Northeastern University

DS 5220 – Lecture 3
Non-Linear Regression

Roi Yehoshua

Agenda

Northeastern University

- Transformable linear regression
- Polynomial regression
- Basis functions
- Model selection and tuning
- Cross-validation
- Regularization
- The bias-variance tradeoff
- Data preprocessing and feature selection

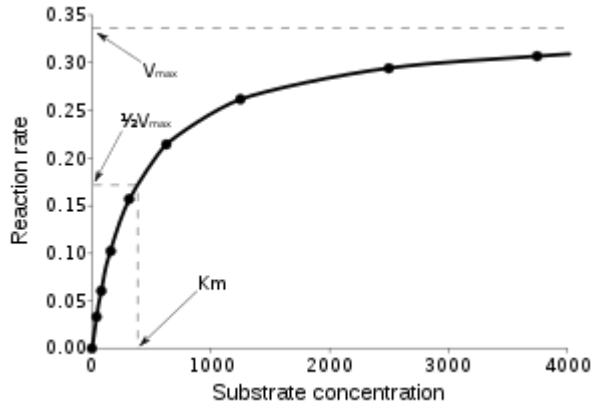
Nonlinear Regression

- In nonlinear regression, the data is modeled by a function which is a nonlinear combination of the model parameters and the independent variables

$$y = f(\mathbf{x}, \mathbf{w})$$

- The function can be an exponential, power, or any other nonlinear function
- For example, the Michaelis–Menten model for enzyme kinetics has two parameters and one independent variable

$$f(x, \mathbf{w}) = \frac{w_1 x}{w_0 + x}$$



Transformable Nonlinear Regression

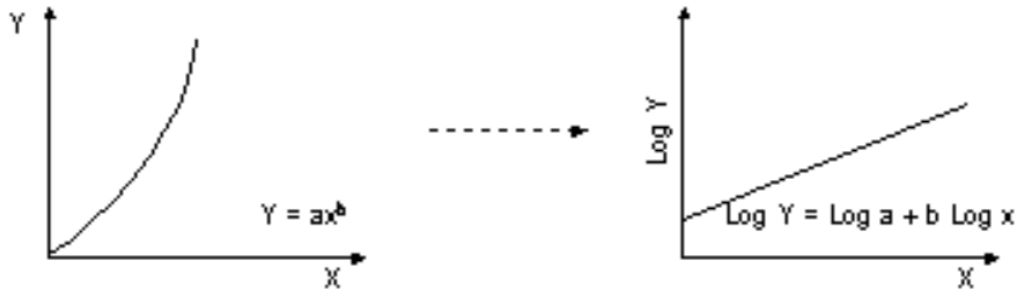
Northeastern University

- There are several non-linear functions that can be transformed into linear functions
- For example, a power function of the form $y = ax^b$ can be linearized by taking logs of both variables

$$Y = \log y = \log(ax^b) = \log a + b \log x = \alpha + \beta X$$

- We first use linear regression between *X* and *Y* to find the coefficients α, β
- Then the parameters of the power curve can be obtained using the transformation:

$$\alpha = \log a, \beta = b \Rightarrow a = e^{\alpha}, b = \beta$$



Transformable Nonlinear Regression

Northeastern University

- Other non-linear transformations:

	Model	Model transformation	Parameters transformation
Power	$y = ax^b$	$Y = \log y, X = \log x$	$\alpha = \log a, \beta = b$
Exponential 1	$y = ae^{bx}$	$Y = \log y, X = x$	$\alpha = \log a, \beta = b$
Exponential 2	$y = ab^x$	$Y = \log y, X = x$	$\alpha = \log a, \beta = \log b$
Logarithmic	$y = \log(ax^b)$	$Y = y, X = \log x$	$\alpha = \log a, \beta = b$
Reciprocal 1	$y = \frac{1}{a+bx}$	$Y = \frac{1}{y}, X = x$	$\alpha = \log a, \beta = b$
Reciprocal 2	$y = a + \frac{b}{1+x}$	$Y = y, X = \frac{1}{1+x}$	$\alpha = a, \beta = b$
Square root	$y = a + b\sqrt{x}$	$Y = y, X = \sqrt{x}$	$\alpha = a, \beta = b$

Basis Functions

Northeastern University

- Linear regression can model non-linear relationships between **x** and *y* by replacing **x** with nonlinear functions of the inputs:

$$h_{\mathbf{w}}(\mathbf{x}) = w_0 + \sum_{j=1}^{d} w_j \phi_j(\mathbf{x})$$

- ϕ_j's are called **basis functions** and can be any type of functions
  - e.g., polynomial, exponential, sigmoidal, and other nonlinear functions
- To simplify notation, we define a dummy basis function $\phi_0(\mathbf{x}) = 1$ so that

$$h_{\mathbf{w}}(\mathbf{x}) = \sum_{j=0}^{d} w_j \phi_j(\mathbf{x}) = \mathbf{w}^T \boldsymbol{\phi}(\mathbf{x})$$

- We can apply the same linear regression methods to find the best-fit vector **w**

Polynomial Regression

Northeastern University

- In polynomial regression the basis functions take the form of $\phi_j(x) = x^j$
- For example, a degree-3 polynomial fits a cubic curve to the data

$$h_{\mathbf{w}}(x) = w_0 + w_1 x + w_2 x^2 + w_3 x^3$$

- If **x** contains more than one feature, we add all the combinations of powers of the features up to a specified degree *d*
- For example, if **x** is a two-dimensional feature vector, and *d* = 2, then

$$h_{\mathbf{w}}(\mathbf{x}) = w_0 + w_1 x_1 + w_2 x_2 + w_3 x_1 x_2 + w_4 x_1^2 + w_5 x_2^2$$

- In this case, the design matrix *X* will be:

$$X = \begin{pmatrix} 1 & x_{11} & x_{12} & x_{11}x_{12} & x_{11}^2 & x_{12}^2 \\ 1 & x_{21} & x_{22} & x_{21}x_{22} & x_{21}^2 & x_{22}^2 \\ \vdots & \vdots & \ddots & \vdots & & \\ 1 & x_{n1} & x_{n2} & x_{n1}x_{n2} & x_{n1}^2 & x_{n2}^2 \end{pmatrix}$$

Polynomial Regression in Scikit-Learn

Northeastern University

- The transformer **PolynomialFeatures** generates a new feature matrix consisting of all polynomial combinations of the features up to the specified degree

```
class sklearn.preprocessing.PolynomialFeatures(degree=2, *, interaction_only=False, include_bias=True, order='C')    [source]
```

- Parameters:
  - interaction_only – if true, only interaction features are produced: features that are products of at most degree *distinct* input features (so not x_1^2 or $x_0x_2^3$, etc.)
  - include_bias – if true (default), include a bias column, the feature in which all polynomial powers are 0 (i.e., a column of ones)

Polynomial Regression in Scikit-Learn

Northeastern University

- Example:

```
X = np.arange(6).reshape(3, 2)
X
```

```
array([[0, 1],
       [2, 3],
       [4, 5]])
```

```
from sklearn.preprocessing import PolynomialFeatures

poly_features = PolynomialFeatures(2)
poly_features.fit_transform(X)
```

```
array([[ 1.,  0.,  1.,  0.,  0.,  1.],
       [ 1.,  2.,  3.,  4.,  6.,  9.],
       [ 1.,  4.,  5., 16., 20., 25.]])
```

Polynomial Regression in Scikit-Learn

Northeastern University

- For example, assume that our target function is a quadratic function of the form

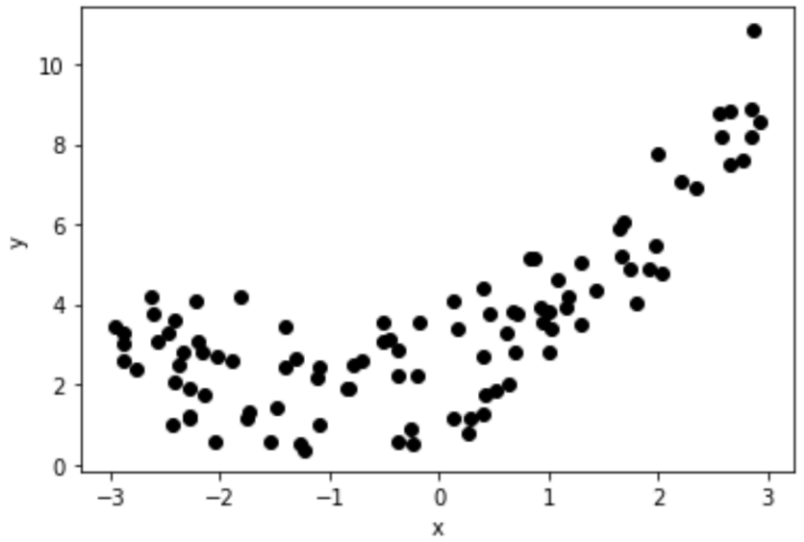
$$y = \frac{x^2}{2} + x + 2$$

- We are given a training set of 100 points of this function in the range [-3, 3]
  - Each point is generated with random Gaussian noise with mean 0 and variance 1

```
def make_data(n=100, err=1.0):
    rng = np.random.RandomState(0)
    x = 6 * rng.rand(n) - 3 # x in [-3,3]
    err = rng.normal(size=n)
    y = 0.5 * x ** 2 + x + 2 + err
    return x, y
```

```
def plot_data(x, y):
    plt.scatter(x, y, color='k')
    plt.xlabel('x')
    plt.ylabel('y')
```

```
x, y = make_data(100)
plot_data(x, y)
```



Polynomial Regression in Scikit-Learn

Northeastern University

- Let's fit a 2-degree polynomial to this data
- First we use PolynomialFeatures to create a new feature matrix that includes x and x^2
  - No need to include an intercept term when using the LinearRegression class

```
poly_features = PolynomialFeatures(degree=2, include_bias=False)
X_poly = poly_features.fit_transform(x.reshape(-1, 1))

X_poly[:5]
```

```
array([[ 0.29288102,  0.08577929],
       [ 1.2911362 ,  1.66703268],
       [ 0.61658026,  0.38017121],
       [ 0.2692991 ,  0.072522  ],
       [-0.4580712 ,  0.20982923]])
```

Polynomial Regression in Scikit-Learn

Northeastern University

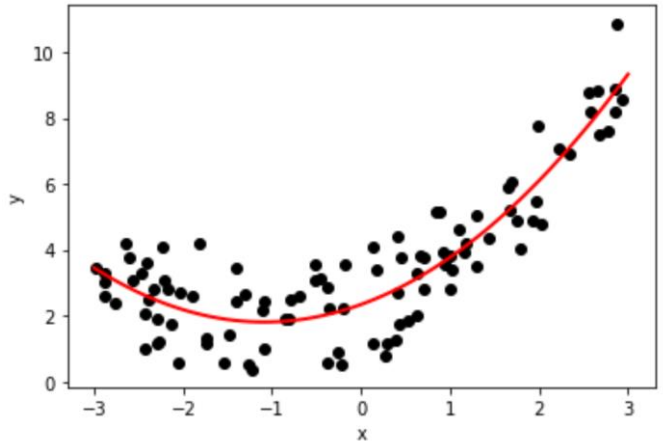
- We can now fit a LinearRegression model to the new feature matrix:

```
from sklearn.linear_model import LinearRegression

reg = LinearRegression()
reg.fit(X_poly, y)
reg.intercept_, reg.coef_
```

```
(2.3405007562628866, array([0.97906552, 0.44978823]))
```

- Not bad: the model estimates $h(x) = 0.45x^2 + 0.98x + 2.34$, when the true function is $y = 0.5x^2 + x + 2$ + Gaussian noise

```
x_test = np.linspace(-3, 3, 100).reshape(-1, 1)
X_test_poly = poly_features.transform(x_test)
y_test = reg.predict(X_test_poly)

plot_data(x, y)
plt.plot(x_test, y_test, lw=2, color='r');
```



Polynomial Regression in Scikit-Learn

Northeastern University

- We usually don't know what is the shape of the target function or even if it is polynomial at all!
- However, we can still try to fit a polynomial to the training data

- **Weierstrass approximation theorem**
  Suppose *f* is a continuous real-valued function defined on the interval [*a*, *b*].
  Then for every $\varepsilon > 0$, there exists a polynomial *p* such that for all $x \in [a, b]$, we have

$$|f(x) - p(x)| < \epsilon$$

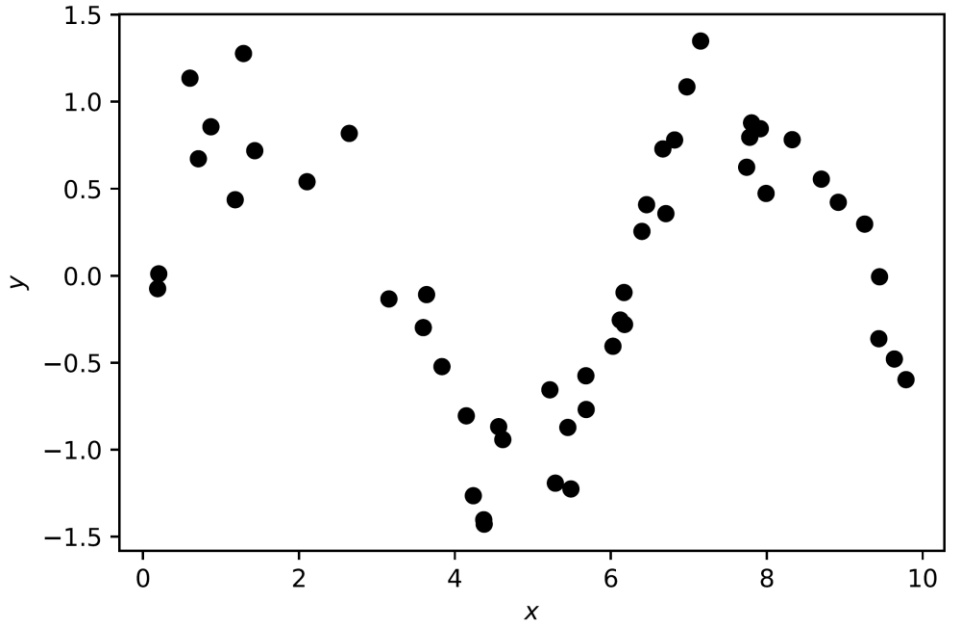
Polynomial Regression in Scikit-Learn

Northeastern University

- For example, assume that we have 50 data points that were generated from the function $f(x) = \sin x$ in the interval [0, 10] with some random noise

```
def make_data(n=50):
    rng = np.random.RandomState(0)
    x = rng.rand(n, 1) * 10
    err = rng.normal(size=(n, 1)) * 0.3
    y = np.sin(x) + err
    return x, y
```

```
x, y = make_data()
plot_data(x, y)
```



Polynomial Regression in Scikit-Learn

Northeastern University

- Let’s try to fit polynomials of various degrees to this data
- To make it easier, we’ll first create a pipeline that combines the polynomial features transformer with the linear regression estimator:

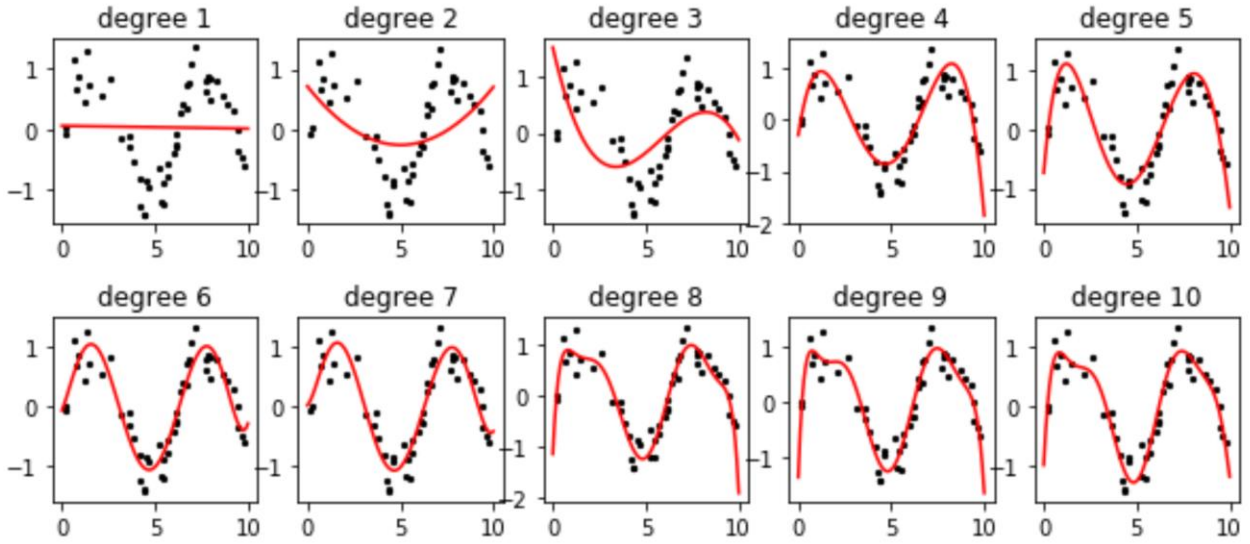
```
from sklearn.pipeline import Pipeline

def PolynomialRegression(degree=2):
    return Pipeline([('polyfeatures', PolynomialFeatures(degree)),
                     ('reg', LinearRegression())])
```

Polynomial Regression in Scikit-Learn

Northeastern University

- Let's fit polynomials of degrees between 1 and 10 to the data and plot the graphs:

```
x_test = np.linspace(0, 10, 100).reshape(-1, 1)

fig, axes = plt.subplots(2, 5, figsize=(10, 4))
plt.subplots_adjust(hspace=0.5)
degree = 1
for ax in axes.flat:
    reg = PolynomialRegression(degree)
    reg.fit(x, y)
    y_test = reg.predict(x_test)
    ax.plot(x_test, y_test, color='r')
    ax.set_title(f'degree {degree}')
    ax.scatter(x, y, color='k', s=5)
    degree += 1
```



Polynomial Regression in Scikit-Learn

Northeastern University

- **Question**: is there a polynomial that passes exactly through all the 50 data points?
- We know from math that for every *n* points in the plane there exists a unique polynomial of degree *n* – 1 that passes through all the points
- So in our case there is a polynomial of degree 49 that perfectly matches the data set
- However, this polynomial may strongly overfit the training data
- Thus, which degree of polynomial we should pick?

Model Selection

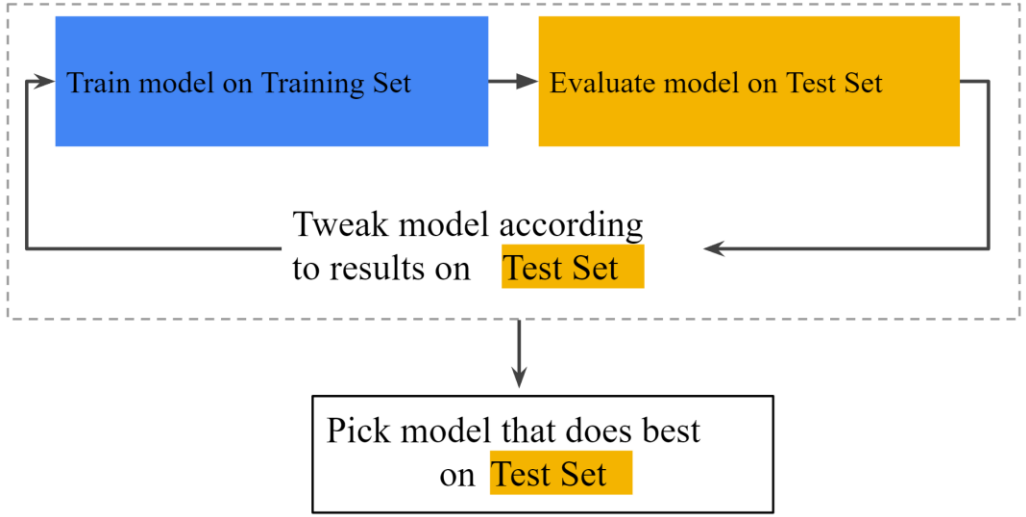
Northeastern University

- **Goal:** choose the best model from a set of candidate models trained on a data set S
- Model selection can be applied both
  - across different types of models (e.g., logistic regression, SVM, KNN, etc.)
  - across models of the same type configured with different hyperparameters
- Typically, we would like to choose the model that performs best on unseen data
  - i.e., has the lowest **generalization error**
- However, other factors may also need to be taken into account, such as:
  - How long it takes to train the model or use the model for predictions
  - How easy is to explain the model's predictions
- **Model tuning** is an iterative process of tweaking the model hyperparameters and the features used to extract the best performance out of any given model

The Holdout Method

Northeastern University

- The data set is split into two disjoint sets, called the **training set** and the **test set**
- Each model is induced from the training set and evaluated on the test set
- We pick the model that has the smallest error on the test set



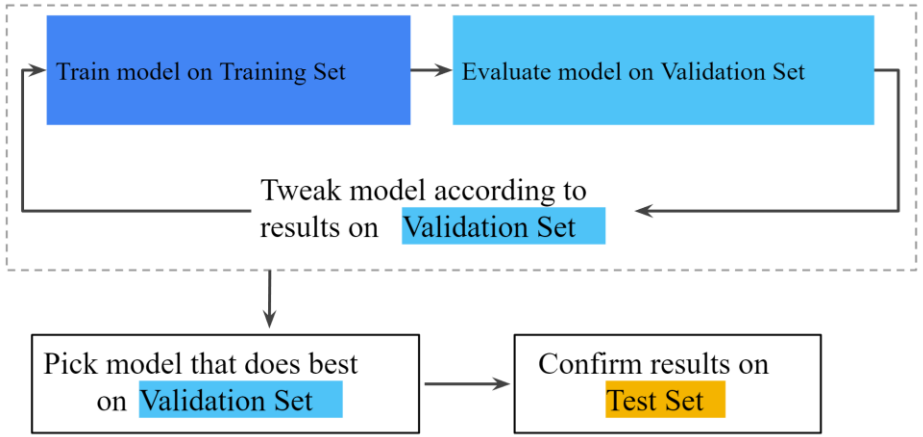
- **Problem**: the test data is being used to make choices about the model parameters
  - can cause the model to overfit to the peculiarities of the specific test set

Validation Set

Northeastern University

- A better approach is to split the data set into three disjoint subsets:

  - Use the validation set to evaluate different configurations of the model
  - Use the test set to double-check the final model that has "passed" the validation set

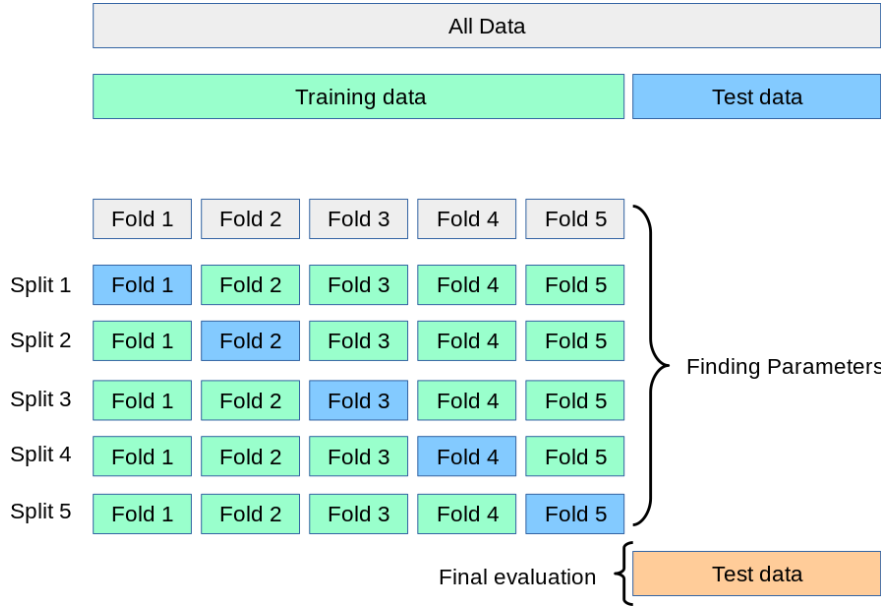


- **Problem**: we lose a decent amount of the training data for validation

k-Fold Cross Validation

Northeastern University

- Randomly split the training set into *k* equal-sized partitions S_1, …, S_k
- For $i = 1, \ldots, k$:
  - Train the model on all the training data except for S_i
  - Test the model on S_i
- The resulting *k* errors are averaged to obtain an estimate of the generalization error



k-Fold Cross Validation

Northeastern University

- **Advantages**:
  - The fraction of data held out in each iteration is typically much smaller than using one validation set (depending on *k*)
  - Provides not only an estimate for the performance of the model, but also a measure of how precise this estimate is (its standard deviation)
- **Disadvantages**:
  - More computationally expensive

Leave-One-Out Cross Validation

Northeastern University

- A special case of *k*-fold cross-validation where $k = n$ (the size of the training set)
- Repeatedly train on all but one of the training examples
  - Test on that one held-out example
- The resulting *n* errors are then averaged to obtain the generalization error

- **Advantage**: utilizing as much data as possible for training
- **Disadvantages**:
  - More computationally expensive
  - The variance of the performance metric tends to be high

Cross Validation in Scikit-Learn

Northeastern University

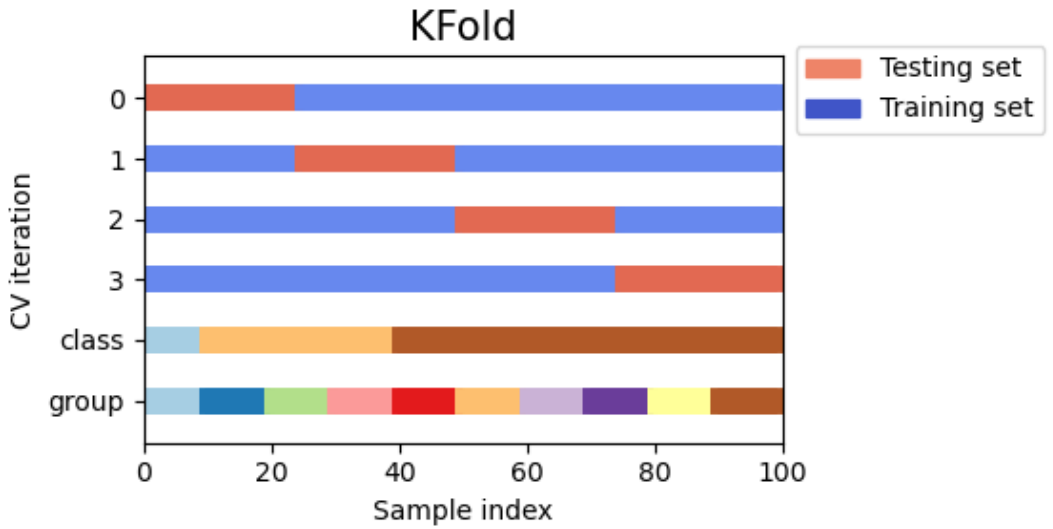
- The method **cross_val_score()** can be used to perform cross-validation

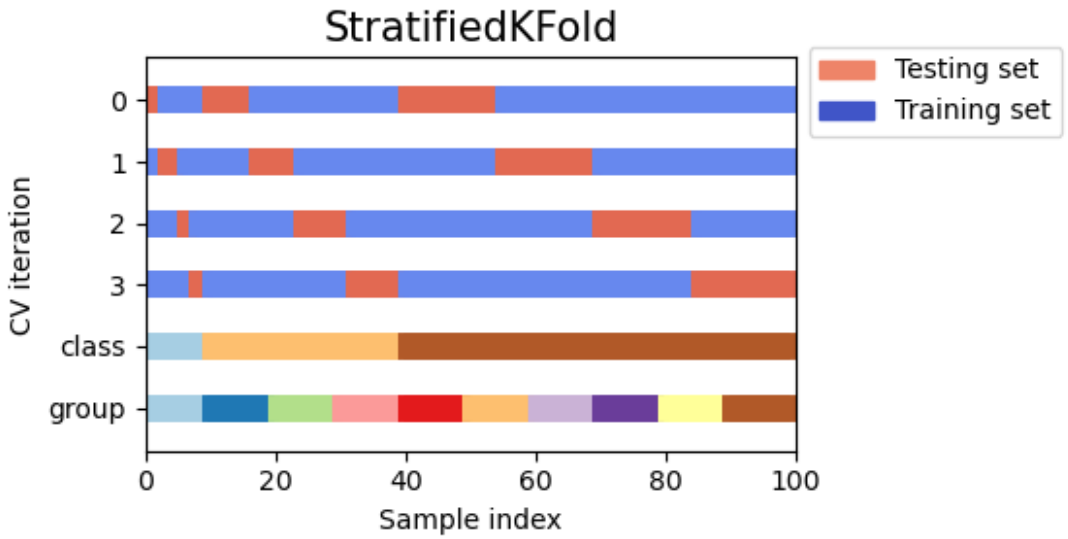
```
sklearn.model_selection.cross_val_score(estimator, X, y=None, *, groups=None, scoring=None, cv=None, n_jobs=None,
verbose=0, fit_params=None, pre_dispatch='2*n_jobs', error_score=nan)                                    [source]
```

- The **cv** argument determines the cross-validation splitting strategy
- Possible inputs for cv are:
  - None: uses the default 5-fold cross validation
  - integer: specifies the number of folds
  - An iterable yielding (train, test) splits as arrays of indices
- For integer/None inputs, if the estimator is a classifier, a **stratified sampling** is used

Stratified Sampling

- Generate folds that preserve the percentage of samples for each class





Cross Validation in Scikit-Learn

Northeastern University

- For example, let's evaluate the 6-degree polynomial by using 5-fold cross validation:

```
from sklearn.model_selection import cross_val_score

reg = PolynomialRegression(degree=6)
scores = cross_val_score(reg, x, y, cv=5)
scores
```

```
array([0.81275184, 0.88752451, 0.58148338, 0.78641255, 0.73651294])
```

- The mean score and the standard deviation are hence given by:

```
print(f'Mean score: {scores.mean():.3f}, std: {scores.std():.3f}')
```

```
Mean score: 0.761, std: 0.102
```

Cross Validation in Scikit-Learn

Northeastern University

- By default, cross-validation uses the score() method of the estimator
- It is possible to change the scoring function by using the **scoring** argument
  - A list of scoring functions can be found here
- All scoring functions follow the convention that higher return values are better than lower return values
- Thus, metrics like RMSE are available as **neg_root_mean_squared_error**
  - which returns the negated value of the metric

```
rmse = -cross_val_score(reg, x, y, scoring='neg_root_mean_squared_error', cv=5)
rmse
```

```
array([0.0578541 , 0.10249305, 0.106592  , 0.13665261, 0.11696349])
```

```
print(f'Mean RMSE: {rmse.mean():.3f}, std: {rmse.std():.3f}')
```

```
Mean RMSE: 0.104, std: 0.026
```

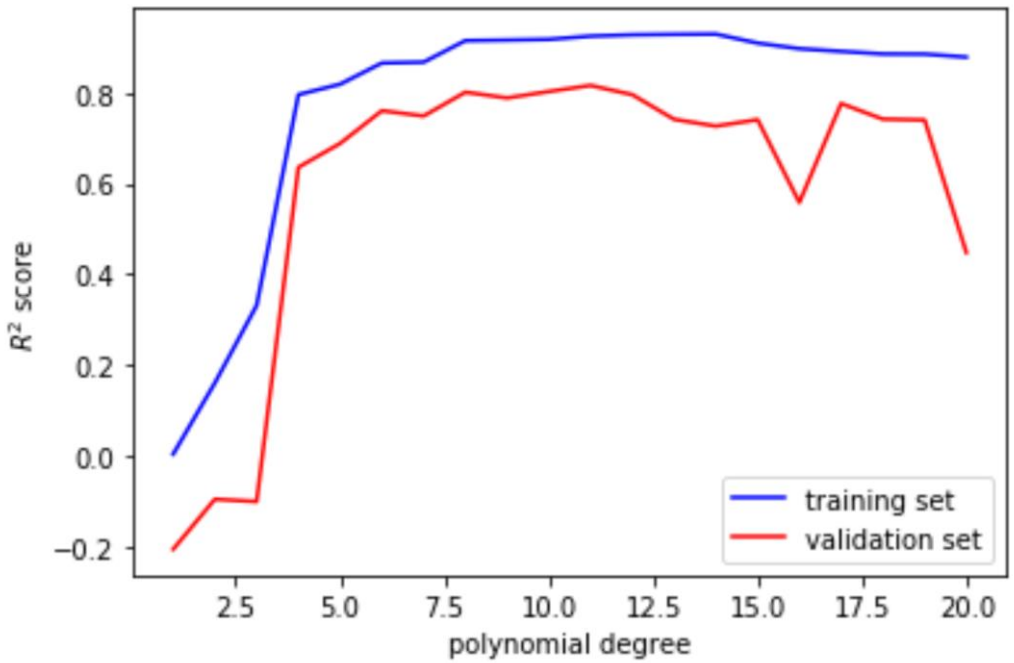
Validation Curve

Northeastern University

- A graph that shows the training and validation scores for varying parameter values

`sklearn.model_selection.` **`validation_curve`**(*estimator, X, y, *, param_name, param_range, groups=None, cv=None, scoring=None, n_jobs=None, pre_dispatch='all', verbose=0, error_score=nan, fit_params=None*) [source]

```
from sklearn.model_selection import validation_curve

degree = np.arange(1, 21)
train_scores, val_scores = validation_curve(
                                PolynomialRegression(),
                                x, y,
                                param_name='polyfeatures__degree',
                                param_range=degree,
                                cv=5
                            )
plt.plot(degree, np.mean(train_scores, axis=1), 'b', label='training set')
plt.plot(degree, np.mean(val_scores, axis=1), 'r', label='validation set')
plt.legend()
plt.xlabel('polynomial degree')
plt.ylabel('$R^2$ score');
```



Other Basis Functions

Northeastern University

- Basis functions other than polynomials are also possible

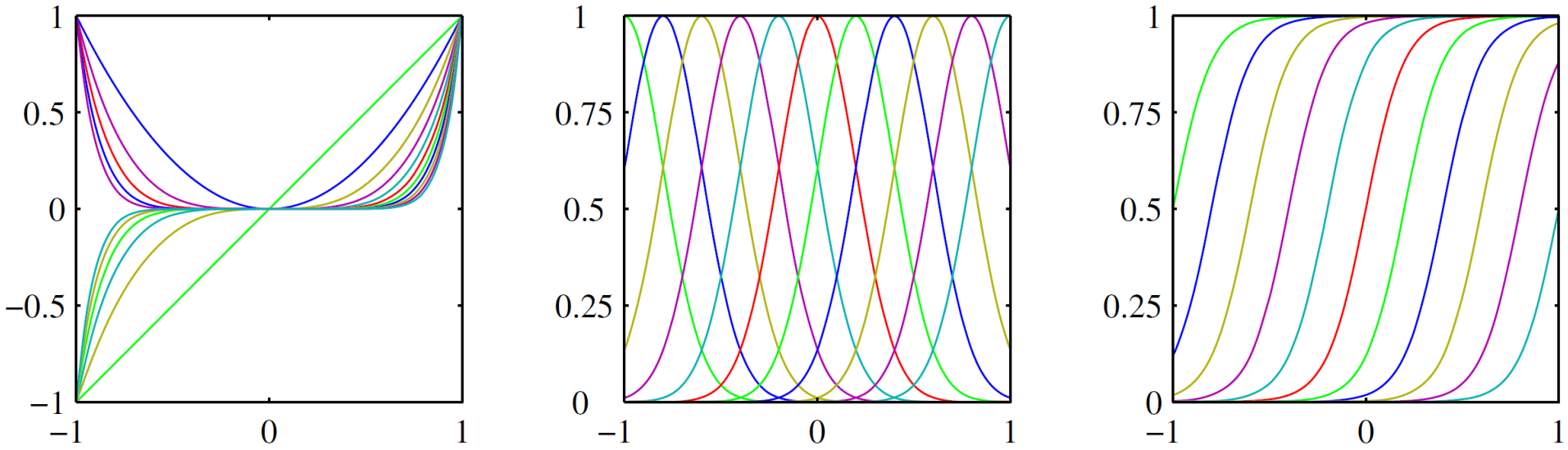
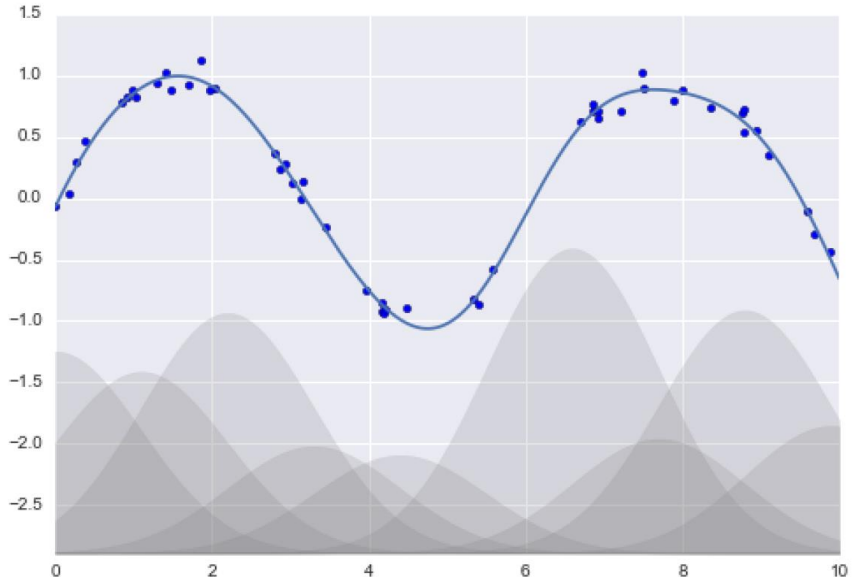


Figure 3.1 Examples of basis functions, showing polynomials on the left, Gaussians of the form (3.4) in the centre, and sigmoidal of the form (3.5) on the right.

Gaussian Basis Functions

Northeastern University

- For example, we can fit a model that is a sum of Gaussian bases:

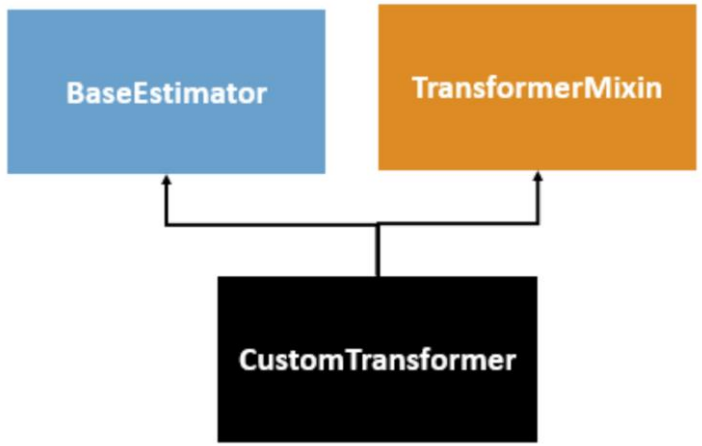


- Gaussian basis functions are not built into Scikit-Learn, but we can write a custom transformer to create them

Custom Transformer

Northeastern University

- Allows you to write your own transformers for tasks such as custom cleanup operations or feature extraction
- A custom transformer is a class that inherits from both **BaseEstimator** and **TransformerMixin**, and implements the methods **fit()** and **transform()**
  - BaseEstimator provides get_params() and set_params() used for hyperparameter tuning
  - TransformerMixin provides fit_transform() which calls fit() and transform()



Gaussian Basis Functions

Northeastern University

- A custom transformer that generates Gaussian basis functions for a given data set X:

```python
from sklearn.base import BaseEstimator, TransformerMixin

class GaussianFeatures(BaseEstimator, TransformerMixin):
    """Uniformly spaced Gaussian features for 1-D input"""

    def __init__(self, n, width_factor=2.0):
        self.n = n
        self.width_factor = width_factor

    def fit(self, X, y=None):
        # Create n centers spread along the data range
        self.centers_ = np.linspace(X.min(), X.max(), self.n)
        self.width_ = self.width_factor * (self.centers_[1] - self.centers_[0])
        return self

    def transform(self, X):
        distances = (X[:, :, np.newaxis] - self.centers_) / self.width_
        new_X = np.exp(-0.5 * np.sum(distances**2, axis=1))
        return new_X
```

Gaussian Basis Functions

Northeastern University

- A pipeline that combines the Gaussian features transformer with linear regression:

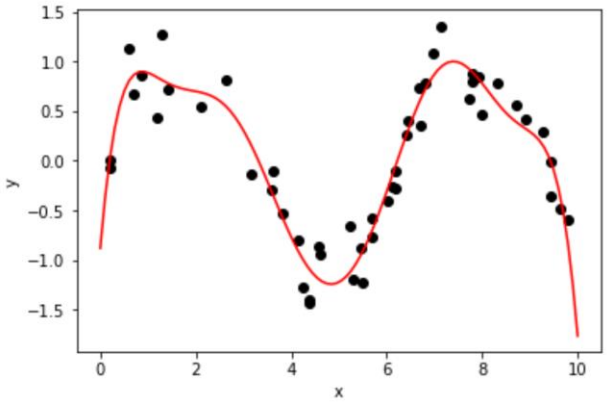
```
def GaussianBasisRegression(n_features=10):
    return Pipeline([('gaussfeatures', GaussianFeatures(n_features)),
                     ('reg', LinearRegression())])
```

- Use the pipeline to find a fit to the data generated by the function $f(x) = \sin x$:

```
gauss_model = GaussianBasisRegression()
gauss_model.fit(x, y)
```

```
Pipeline(steps=[('gaussfeatures', GaussianFeatures(n=10)),
                ('reg', LinearRegression())])
```

```
x_test = np.linspace(0, 10, 100).reshape(-1, 1)
y_test = gauss_model.predict(x_test)
plot_data(x, y)
plt.plot(x_test, y_test, color='r');
```



Regularization

Northeastern University

- A technique to prevent overfitting by penalizing complex models
- Changes the cost function of a model to take into account both the errors on the training set and the complexity of the model:

$$Cost(h) = TrainingError(h) + \lambda\, Complexity(h)$$

  - The hyperparameter λ controls how much you want to regularize the model
- For linear models, the complexity can be specified as the size of the feature weights:

$$Complexity(h) = L_q(\mathbf{w}) = \sum_{j=0}^{d} |w_j|^q$$

  - with q = 1, we have **L1 regularization**, which minimizes the sum of absolute values
  - with q = 2, we have **L2 regularization**, which minimizes the sum of squares
- Often the bias weight w_0 is not regularized

Ridge Regression

Northeastern University

- **Ridge regression** is the L2 regularized version of linear regression
- The cost function in ridge regression is defined as:

$$J(\mathbf{w}) = \sum_{i=1}^{n}(h_{\mathbf{w}}(\mathbf{x}_i) - y_i)^2 + \lambda \sum_{j=0}^{d} w_j^2 = ||X\mathbf{w} - \mathbf{y}||_2^2 + \lambda||\mathbf{w}||_2^2$$

- Its closed-form solution is:

$$\mathbf{w}^* = \left(X^T X + \lambda I\right)^{-1} X^T \mathbf{y}$$

  - you will prove this in the homework assignment

- For both the closed-form solution and gradient descent, it is important to normalize the data as regularization is sensitive to the scale of the input features

Ridge Regression in Scikit-Learn

Northeastern University

- To perform ridge regression using a closed-form solution, use the **Ridge** class:

> *class* `sklearn.linear_model.` **`Ridge`**(*alpha=1.0, *, fit_intercept=True, normalize=False, copy_X=True, max_iter=None, tol=0.001, solver='auto', random_state=None*) [source]

  - The alpha parameter specifies the regularization strength
- For SGD solution, use **SGDRegressor** with penalty='l2' (the default)

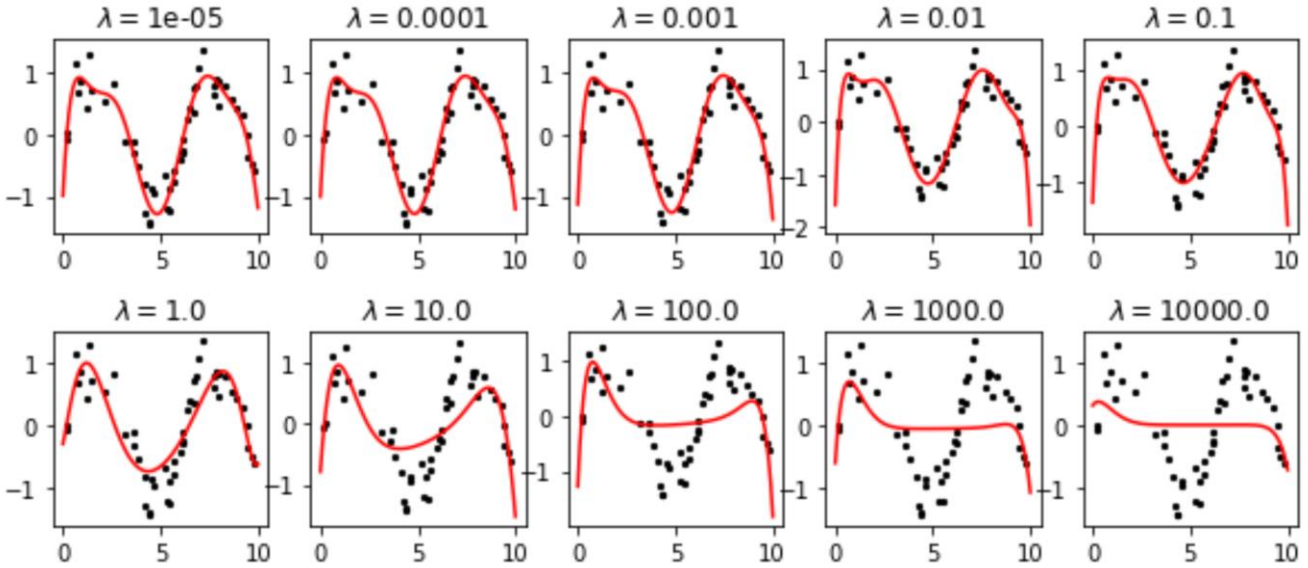
- Let's define a pipeline that combines PolynomialFeatures with Ridge regression:

```
from sklearn.linear_model import Ridge

def RidgePolynomialRegression(degree=2, alpha=1):
    return Pipeline([('scaler', StandardScaler()),
                     ('polyfeatures', PolynomialFeatures(degree)),
                     ('ridge', Ridge(alpha))])
```

Ridge Regression in Scikit-Learn

- We now use ridge regression to fit a polynomial of degree 10 to the function $f(x) = \sin x$ in the interval [0, 10] using various regularization parameters:

```
fig, axes = plt.subplots(2, 5, figsize=(10, 4))
plt.subplots_adjust(hspace=0.5)
alpha = 0.00001

for ax in axes.flat:
    reg = RidgePolynomialRegression(degree=10, alpha=alpha)
    reg.fit(x, y)
    y_test = reg.predict(x_test)
    ax.plot(x_test, y_test, color='r')
    ax.set_title(f'$\lambda = ${alpha}')
    ax.scatter(x, y, color='k', s=5)
    alpha *= 10
```



- For $\lambda = 1.0$, the overfitting is suppressed and we obtain a much closer representation of the function $\sin x$
- However, if we choose too large a value for λ, we again obtain a poor fit

Ridge Regression in Scikit-Learn

Northeastern University

- The coefficients of the fitted polynomials are:

```
alpha = 0.00001
for i in range(10):
    reg = RidgePolynomialRegression(degree=10, alpha=alpha)
    reg.fit(X, y)
    coef = reg.named_steps['ridge'].coef_[1:]
    print(coef)
    alpha *= 10
```

```
[ 2.45811336  5.07749379 -5.06399356 -5.79086794  3.87586275  3.10964014
 -1.32463557 -0.80719897  0.15995294  0.07845666]
[ 2.45003426  5.0488804  -5.01975307 -5.71852554  3.81284185  3.0418889
 -1.29246886 -0.78026437  0.15456612  0.07460127]
[ 2.3796623   4.80023684 -4.63603951 -5.09208282  3.26768116  2.45642586
 -1.01466256 -0.5478251   0.10809414  0.04136073]
[ 2.08199375  3.77382427 -3.08373902 -2.60115876  1.12473195  0.18157403
  0.0577718   0.34173187 -0.06908624 -0.08451899]
[ 1.6461383   2.54407891 -1.5762117  -0.6575335  -0.26159415 -0.99333451
  0.52897486  0.64461632 -0.12137349 -0.11184798]
[ 0.96701995  1.07458368 -0.31639155  0.14799578 -0.23867893 -0.25638155
 -0.00456362 -0.00477542  0.02346869  0.01398101]
[ 0.38811717  0.32322951  0.04661653  0.18997522 -0.06103618  0.04841547
 -0.08149436 -0.08424857  0.01752565  0.0108085 ]
[ 0.07302939  0.06003444  0.02638145  0.05437841  0.00729324  0.04498149
 -0.00546776  0.02389135 -0.01319033 -0.01824813]
[ 0.00767524  0.00982117  0.00154291  0.01325401 -0.00323177  0.01851608
 -0.00847988  0.02091131 -0.00443604 -0.01086385]
[ 0.00051447  0.00131032 -0.00041546  0.00207992 -0.00154887  0.00329475
 -0.00313205  0.00397903 -0.00304347 -0.00293207]
```

- Regularization has the desired effect of reducing the magnitude of the coefficients

Lasso Regression

Northeastern University

- Lasso regression is the L1 regularized version of linear regression
- The cost function in lasso regression is defined as:

$$J(\mathbf{w}) = \sum_{i=1}^{n}(h_{\mathbf{w}}(\mathbf{x}_i) - y_i)^2 + \lambda \sum_{j=0}^{d} |w_j| = ||X\mathbf{w} - \mathbf{y}||_2^2 + \lambda||\mathbf{w}||_1$$

- There is no closed-form solution, but can optimize using subgradient descent
- The cost function is not differentiable at $w_j = 0$, but gradient descent still works fine if a **subgradient vector** is used instead when any $w_j = 0$
  - A subgradient vector is an intermediate vector between the gradient vectors around that point

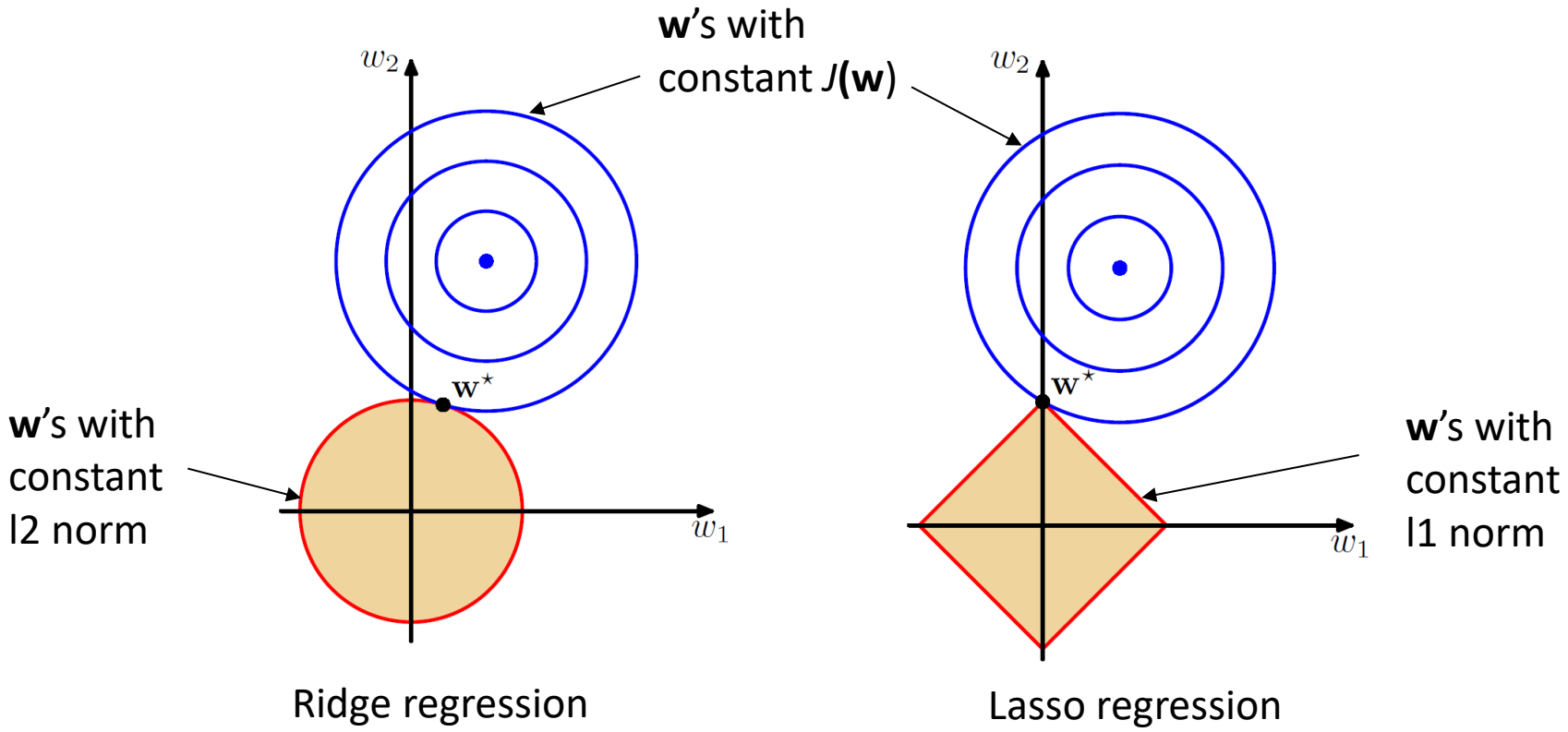
Lasso vs. Ridge Regression

Northeastern University

- Lasso results in sparse solutions (a vector with more zero coordinates)
  - The derivative of the L1 penalty with respect to each weight w_i is constant while the derivative of L2 penalty is $2w_i$
  - This causes the weights to drop to exactly 0 where possible, while L2 regularization encourages weights to be small, but doesn't force them to exactly 0
- Good for high-dimensional problems
  - Don’t have to store all coordinates
  - Interpretable solution

Lasso vs. Ridge Regression

- Geometrical interpretation



Lasso Regression in Scikit-Learn

Northeastern University

- The **Lasso** class implements a lasso regression:

> *class* `sklearn.linear_model.` **`Lasso`**(*alpha=1.0, *, fit_intercept=True, normalize=False, precompute=False, copy_X=True, max_iter=1000, tol=0.0001, warm_start=False, positive=False, random_state=None, selection='cyclic'*) [source]

  - The alpha parameter specifies the regularization strength
  - Uses a coordinate descent solver
- For a SGD solution, use **SGDRegressor** with penalty='l1'
- Let's define a pipeline that combines PolynomialFeatures with Lasso regression:

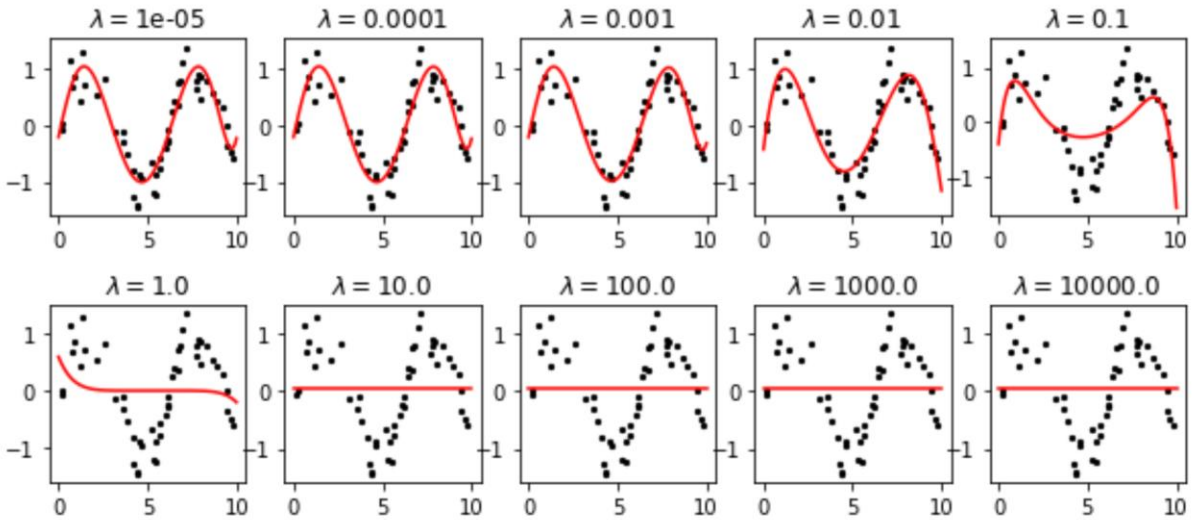
```
from sklearn.linear_model import Lasso

def LassoPolynomialRegression(degree=2, alpha=1):
    return Pipeline([('scaler', StandardScaler()),
                     ('polyfeatures', PolynomialFeatures(degree)),
                     ('lasso', Lasso(alpha))])
```

Lasso Regression in Scikit-Learn

Northeastern University

- Using lasso regression to fit a polynomial of degree 10 to the function sin*x*:

```
fig, axes = plt.subplots(2, 5, figsize=(10, 4))
plt.subplots_adjust(hspace=0.5)
alpha = 0.00001

for ax in axes.flat:
    reg = LassoPolynomialRegression(degree=10, alpha=alpha)
    reg.fit(x, y)
    y_test = reg.predict(x_test)
    ax.plot(x_test, y_test, color='r')
    ax.set_title(f'$\lambda = ${alpha}')
    ax.scatter(x, y, color='k', s=5)
    alpha *= 10
```



- In this case a much smaller λ was needed to get a good representation of the function sin*x*

Elastic Net

Northeastern University

- **Elastic net** is a middle ground between ridge regression and lasso regression
- The regularization term linearly combines the L1 and L2 penalties
- The elastic net cost function is:

$$J(\mathbf{w}) = \sum_{i=1}^{n}(h_{\mathbf{w}}(\mathbf{x}_i) - y_i)^2 + r\lambda\sum_{j=0}^{d}|w_j| + \frac{1-r}{2}\lambda\sum_{j=0}^{d}w_j^2$$

  - The mix ratio $0 \leq r \leq 1$ is a hyperparameter

Elastic Net in Scikit-Learn

Northeastern University

- The **ElasticNet** class implements the elastic net algorithm:

> *class* `sklearn.linear_model.` **`ElasticNet`**(*alpha=1.0, *, l1_ratio=0.5, fit_intercept=True, normalize=False, precompute=False, max_iter=1000, copy_X=True, tol=0.0001, warm_start=False, positive=False, random_state=None, selection='cyclic'*) [source]

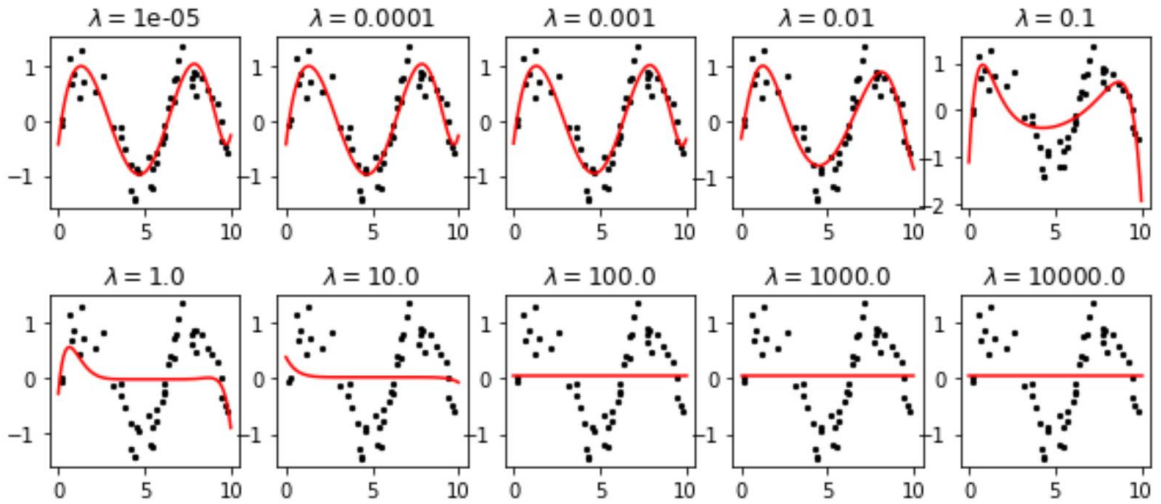
  - l1_ratio is the ratio between the L1 penalty and the L2 penalty
    - For l1_ratio = 0, the penalty is L2 penalty
    - For l1_ratio = 1, it is an L1 penalty
    - Default is 0.5
- Let's define a pipeline that combines PolynomialFeatures with elastic net:

```
from sklearn.linear_model import ElasticNet

def ElasticNetPolynomialRegression(degree=2, alpha=1):
    return Pipeline([('scaler', StandardScaler()),
                     ('polyfeatures', PolynomialFeatures(degree)),
                     ('elasticnet', ElasticNet(alpha))])
```

Elastic Net in Scikit-Learn

- Using elastic net to fit a polynomial of degree 10 to the function sin*x*:

```
fig, axes = plt.subplots(2, 5, figsize=(10, 4))
plt.subplots_adjust(hspace=0.5)
alpha = 0.00001

for ax in axes.flat:
    reg = ElasticNetPolynomialRegression(degree=10, alpha=alpha)
    reg.fit(x, y)
    y_test = reg.predict(x_test)
    ax.plot(x_test, y_test, color='r')
    ax.set_title(f'$\lambda = ${alpha}')
    ax.scatter(x, y, color='k', s=5)
    alpha *= 10
```



- The results are similar to the Lasso regression

Which Regression to Choose?

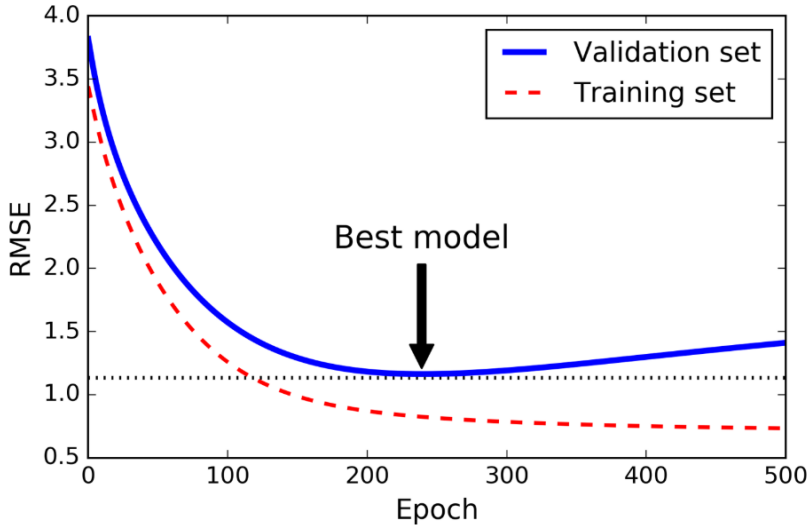
Northeastern University

- It’s almost always preferable to have some regularization, so generally you should avoid plain linear regression
- Ridge is a good default, but if you suspect that only a few features are actually useful, you should prefer lasso or elastic net
- In general, elastic net is preferred over lasso since lasso may behave erratically when several features are strongly correlated

Early Stopping

Northeastern University

- Early stopping is another way to regularize iterative learning algorithms such as GD
- We stop the training as soon as the validation error reaches the minimum:



- With SGD and mini-batch GD, the curves aren't so smooth, so we typically stop the training after the validation error hasn't improved for some number of iterations

Early Stopping

Northeastern University

- To enable early stopping set the early_stopping argument in SGDRegressor to True
  - It's False by default
- If set to True, the training will terminate when validation score is not improving by at least *tol* (defaults to 0.001) for *n_iter_no_change* consecutive epochs (defaults to 5)

class `sklearn.linear_model.` **`SGDRegressor`**(*loss='squared_loss', *, penalty='l2', alpha=0.0001, l1_ratio=0.15, fit_intercept=True, max_iter=1000, tol=0.001, shuffle=True, verbose=0, epsilon=0.1, random_state=None, learning_rate='invscaling', eta0=0.01, power_t=0.25, early_stopping=False, validation_fraction=0.1, n_iter_no_change=5, warm_start=False, average=False*) [source]

Hyperparameter Tuning

Northeastern University

- There are several approaches to hyperparameter tuning:
  - **Manual**: select hyperparameters based on intuition/experience/guessing, train the model and score on the validation set. Repeat until you are satisfied with the results.
  - **Grid search**: set up a grid of hyperparameter values and train the model on each combination of values and score on the validation set
  - **Random search**: set up a grid of hyperparameter values and select *random* combinations of values to train the model and score on the validation set
  - **Automated hyperparameter tuning**: use methods such as Bayesian optimization or evolutionary algorithms to conduct a guided search for the best hyperparameters

Grid Search

Northeastern University

- GridSearchCV performs an exhaustive search over specified parameter values

> *class* `sklearn.model_selection.` **`GridSearchCV`**(*estimator, param_grid, *, scoring=None, n_jobs=None, refit=True, cv=None, verbose=0, pre_dispatch='2*n_jobs', error_score=nan, return_train_score=False*) [source]

Argument	Description
estimator	An object that implements the scikit-learn estimator interface
param_grid	Dictionary with parameter names as keys and lists of parameter settings to try as values
scoring	Strategy to evaluate the performance of the cross-validated model on the test set
n_jobs	Number of jobs to run in parallel (-1 means all processors)
refit	Refit the estimator using the best found parameters on the whole dataset (default True)
cv	Determines the cross-validation splitting strategy

- The search is invoked by calling the **fit()** method of the grid search object

Grid Search Example

Northeastern University

- For example, we can use grid search to find the optimal polynomial model:

```
from sklearn.model_selection import GridSearchCV

param_grid = {
    'polyfeatures__degree': np.arange(1, 11),
    'reg__alpha': np.logspace(-3, 2, num=6)
}

grid = GridSearchCV(RidgePolynomialRegression(), param_grid, n_jobs=-1)
grid.fit(x, y)
```

```
GridSearchCV(estimator=Pipeline(steps=[('scaler', StandardScaler()),
                                       ('polyfeatures', PolynomialFeatures()),
                                       ('reg', Ridge(alpha=1))]),
             n_jobs=-1,
             param_grid={'polyfeatures__degree': array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10]),
                         'reg__alpha': array([1.e-03, 1.e-02, 1.e-01, 1.e+00, 1.e+01, 1.e+02])})
```

Grid Search Example

Northeastern University

- To get the best combination of hyperparameters and the best estimator use:

```
grid.best_params_
```

```
{'polyfeatures__degree': 8, 'ridge__alpha': 0.01}
```

```
grid.best_estimator_
```

```
Pipeline(steps=[('scaler', StandardScaler()),
                ('polyfeatures', PolynomialFeatures(degree=8)),
                ('ridge', Ridge(alpha=0.01))])
```

Grid Search Example

Northeastern University

- You can also inspect the cross-validation scores for each combination of parameters:

```
results = grid.cv_results_
for mean_score, params in zip(results['mean_test_score'], results['params']):
    print(mean_score, params)
```

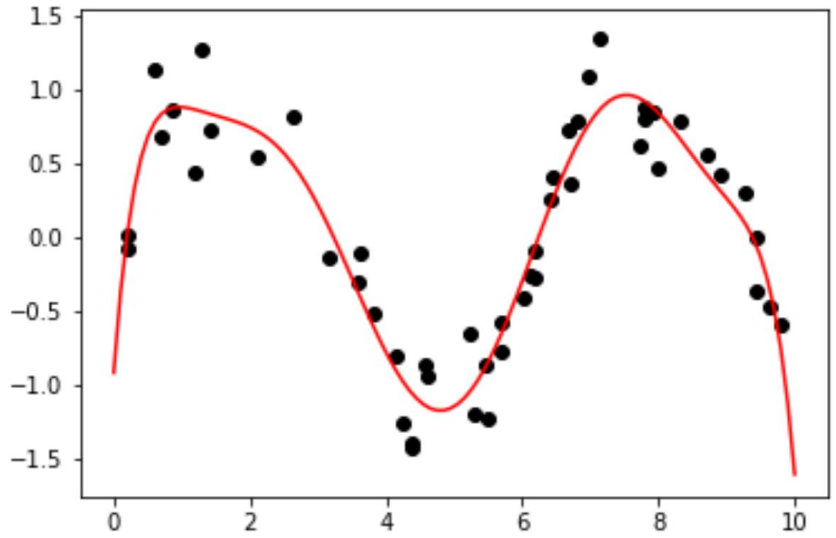
```
-0.2071340351460375 {'polyfeatures__degree': 1, 'ridge__alpha': 0.001}
-0.20712874166460282 {'polyfeatures__degree': 1, 'ridge__alpha': 0.01}
-0.20707595277841864 {'polyfeatures__degree': 1, 'ridge__alpha': 0.1}
-0.20656226484509688 {'polyfeatures__degree': 1, 'ridge__alpha': 1.0}
-0.20253791642483213 {'polyfeatures__degree': 1, 'ridge__alpha': 10.0}
-0.19172852234699103 {'polyfeatures__degree': 1, 'ridge__alpha': 100.0}
-0.09655454408679494 {'polyfeatures__degree': 2, 'ridge__alpha': 0.001}
-0.09650656533407273 {'polyfeatures__degree': 2, 'ridge__alpha': 0.01}
-0.09602992937727488 {'polyfeatures__degree': 2, 'ridge__alpha': 0.1}
-0.09156456448900115 {'polyfeatures__degree': 2, 'ridge__alpha': 1.0}
-0.067127790634579 {'polyfeatures__degree': 2, 'ridge__alpha': 10.0}
-0.09769715006786987 {'polyfeatures__degree': 2, 'ridge__alpha': 100.0}
-0.10177925841315434 {'polyfeatures__degree': 3, 'ridge__alpha': 0.001}
-0.10121515020289167 {'polyfeatures__degree': 3, 'ridge__alpha': 0.01}
-0.09579195653111776 {'polyfeatures__degree': 3, 'ridge__alpha': 0.1}
-0.05796347643045154 {'polyfeatures__degree': 3, 'ridge__alpha': 1.0}
-0.01773291424940926 {'polyfeatures__degree': 3, 'ridge__alpha': 10.0}
-0.07652254551365123 {'polyfeatures__degree': 3, 'ridge__alpha': 100.0}
```

Grid Search Example

Northeastern University

- GridSearchCV also supports the predict() and score() methods if they are implemented in the estimator used
- For example, we can use grid.predict() to plot the best polynomial that was found:

```
y_test = grid.predict(x_test)
plt.plot(x_test, y_test, color='r')
plt.scatter(x, y, color='k');
```



Random Search

Northeastern University

- When the hyperparameter space is large, you can use **RandomizedSearchCV** instead

> *class* `sklearn.model_selection.` **`RandomizedSearchCV`**(*estimator, param_distributions, *, n_iter=10, scoring=None, n_jobs=None, refit=True, cv=None, verbose=0, pre_dispatch='2*n_jobs', random_state=None, error_score=nan, return_train_score=False*) [source]

- In contrast to GridSearchCV, not all parameter values are tried out, but rather a fixed number of parameter settings is sampled from the specified distributions
- The number of parameter settings that are tried is given by **n_iter**
- If all parameters are presented as a list, sampling without replacement is performed
- If at least one parameter is given as a distribution, sampling with replacement is used

Random Search Example

Northeastern University

- For example, let's use random search to find the optimal polynomial model:

```
from sklearn.model_selection import RandomizedSearchCV

param_grid = {
    'polyfeatures__degree': np.arange(1, 11),
    'ridge__alpha': np.logspace(-3, 2, num=6)
}

random_search = RandomizedSearchCV(RidgePolynomialRegression(), param_grid, n_jobs=-1)
random_search.fit(x, y)
```

```
RandomizedSearchCV(estimator=Pipeline(steps=[('scaler', StandardScaler()),
                                             ('polyfeatures',
                                              PolynomialFeatures()),
                                             ('ridge', Ridge(alpha=1))]),
                   n_jobs=-1,
                   param_distributions={'polyfeatures__degree': array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10]),
                                        'ridge__alpha': array([1.e-03, 1.e-02, 1.e-01, 1.e+00, 1.e+01, 1.e+02])})
```

```
random_search.best_params_
```

```
{'ridge__alpha': 0.001, 'polyfeatures__degree': 7}
```

Random Search Example

Northeastern University

- We can examine which combinations of parameters were tested by using the cv_results_ attribute:

```
random_search.cv_results_['params']
```

```
[{'ridge__alpha': 0.001, 'polyfeatures__degree': 3},
 {'ridge__alpha': 0.01, 'polyfeatures__degree': 3},
 {'ridge__alpha': 0.1, 'polyfeatures__degree': 5},
 {'ridge__alpha': 0.001, 'polyfeatures__degree': 7},
 {'ridge__alpha': 0.01, 'polyfeatures__degree': 5},
 {'ridge__alpha': 0.1, 'polyfeatures__degree': 9},
 {'ridge__alpha': 10.0, 'polyfeatures__degree': 4},
 {'ridge__alpha': 1.0, 'polyfeatures__degree': 6},
 {'ridge__alpha': 0.1, 'polyfeatures__degree': 8},
 {'ridge__alpha': 0.1, 'polyfeatures__degree': 7}]
```

Regression Summary

Northeastern University

Algorithm	Scikit-Learn Class
Linear regression (closed form)	LinearRegression
Linear regression (SGD)	SGDRegressor
Polynomial regression (closed form)	PolynomialFeatures + LinearRegression
Polynomial regression (SGD)	PolynomialFeatures + SGDRegressor
Ridge regression (closed form)	Ridge
Ridge regression (SGD)	SGDRegressor (penalty='l2')
Lasso regression (closed form)	Lasso
Lasso regression (SGD)	SGDRegressor (penalty='l1')
Elastic net (closed form)	ElasticNet
Elastic net (SGD)	SGDRegressor (penalty='elasticnet')

Beyond Linear Regression

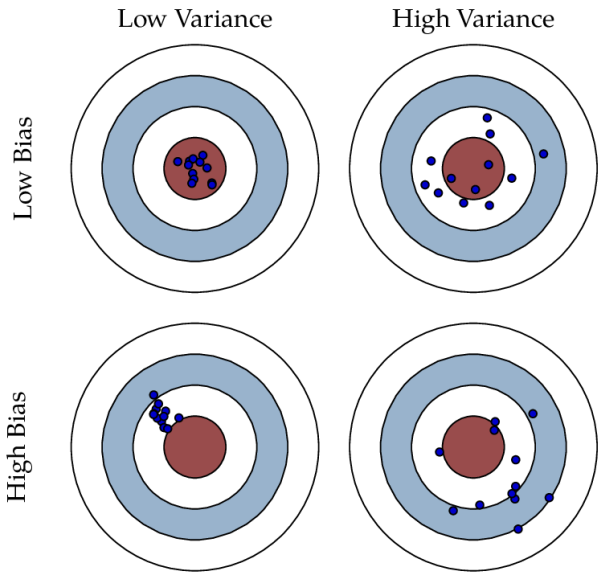
Northeastern University

- Linear regression is sometimes not appropriate, especially for non-linear models of high complexity
- There are other regression techniques suitable for cases where linear regression doesn't work well
- These include decision trees, random forests, SVMs (support vector machines) and neural networks
- These models will be discussed later in the course

The Bias-Variance Tradeoff

Northeastern University

- A formal method for analyzing the prediction errors of a model
- The generalization error of a model can be decomposed into 3 components:
  - Intrinsic noise in the target variable
    - e.g., due to noise in the measurements
    - This type of error cannot be avoided
  - **Bias** in the model
    - Difference between the model's predictions and the targets
  - Sensitivity of the model to the specific training set used
    - or **variance** of the model estimates across different training sets



The Bias-Variance Tradeoff

Northeastern University

- Assume that the points in the data set are sampled from the same distribution with

$$y = f(\mathbf{x}) + \epsilon$$

- The noise ε satisfies: $\mathbb{E}[\epsilon] = 0, \mathrm{Var}(\epsilon) = \sigma^2$
- Our goal is to compute $f(\mathbf{x})$
- By using a training set D, we obtain an estimate $h_D(\mathbf{x})$
- $h_D(\mathbf{x})$ is a random variable function that depends on the choice of the training set
- The squared loss for any given sample point **x** in the test set is:

$$(y - h_D(\mathbf{x}))^2$$

- Since the squared loss depends on the particular training set D, we take its average over the possible training sets: $\mathbb{E}_D\left[(y - h_D(\mathbf{x})^2\right]$
- Our goal is to minimize the mean squared loss for all the samples in the test set

The Bias-Variance Tradeoff

Northeastern University

- We can now write:

$$
\begin{aligned}
\text{Test MSE} &= \mathbb{E}_D\left[(y - h_D(\mathbf{x})^2\right] \\
&= \mathbb{E}_D\left[(f(\mathbf{x}) + \epsilon - h_D(\mathbf{x}))^2\right] \\
&= \mathbb{E}_D\left[(f(\mathbf{x}) - h_D(\mathbf{x}))^2 + \epsilon^2 - 2(f(\mathbf{x}) - h_D(\mathbf{x}))\epsilon\right] \\
&= \mathbb{E}_D\left[(f(\mathbf{x}) - h_D(\mathbf{x}))^2\right] + \mathbb{E}_D[\epsilon^2] - 2\,\mathbb{E}_D[(f(\mathbf{x}) - h_D(\mathbf{x}))\epsilon] \\
&= \mathbb{E}_D\left[(f(\mathbf{x}) - h_D(\mathbf{x}))^2\right] + \mathbb{E}_D[\epsilon^2] \\
&= \left(\mathbb{E}_D[f(\mathbf{x}) - h_D(\mathbf{x})]\right)^2 + \mathrm{Var}_D\left(f(\mathbf{x}) - h_D(\mathbf{x})\right) + \sigma^2 \\
&= \left(\text{Bias } h_D(\mathbf{x})\right)^2 + \mathrm{Var}_D(h_D(\mathbf{x})) + \sigma^2
\end{aligned}
$$

This term is 0, since ε is independent of h_D and $E[\varepsilon] = 0$

$Var[X] = E[X^2] - E[X]^2$

- Conclusion:

MSE = $(\text{bias})^2$ + variance + noise

The Bias-Variance Tradeoff

Northeastern University

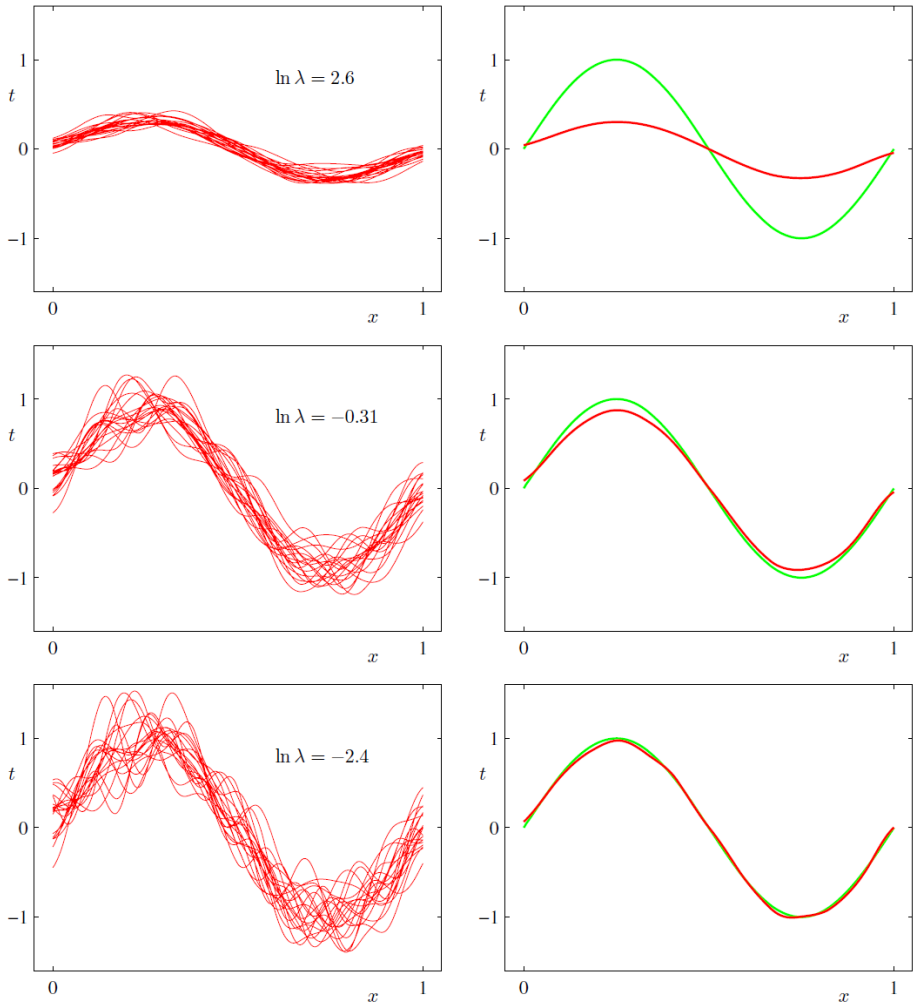
- There is nothing we can do about the term σ^2, as we cannot predict the noise ε
- The bias term is due to **underfitting**: on average *h* doesn't predict well *f*
- The variance term is due to **overfitting**: *h* is too closely related to the training examples, and varies a lot with the choice of our training set
- To sum up, we can understand our MSE as follows:

High bias	$\Leftrightarrow$	Underfitting
High variance	$\Leftrightarrow$	Overfitting
Large σ^2	$\Leftrightarrow$	Noisy data

- There is a **tradeoff between bias and variance**
  - Very flexible models have low bias and high variance
  - Relatively rigid models have high bias and low variance
- The optimal model is the one that best balances between bias and variance

The Bias-Variance Tradeoff

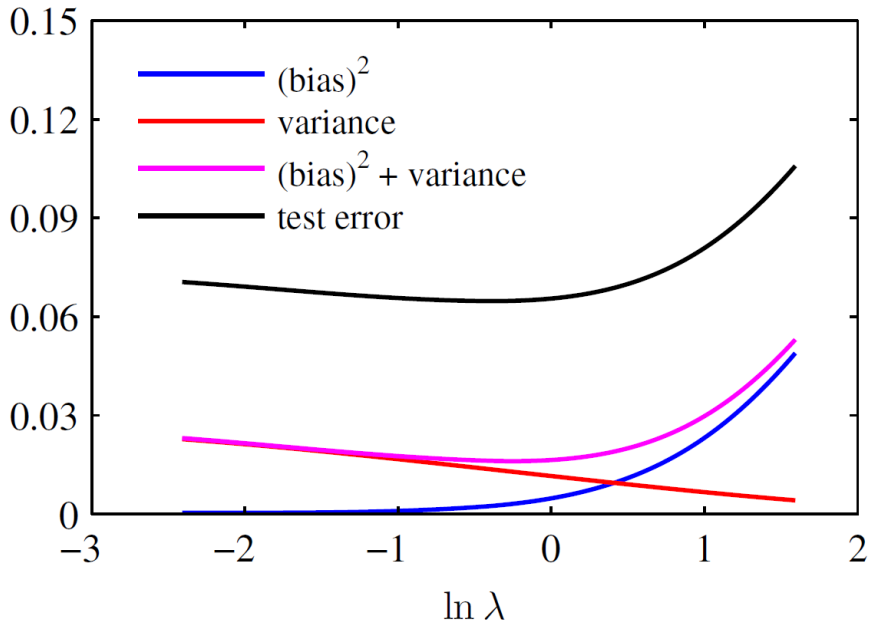
- Illustration of the dependence of the bias and variance on model complexity
- The target function is $\sin 2\pi x$ in [0, 1]
- 100 data sets were used for training, each having 25 data points
- The model consists of 24 Gaussian basis functions
- Left column shows the result of fitting the model to the data sets for different values of the regularization parameter λ
- Right column shows the corresponding average of the 100 fits (red) with the target function (green)
- Example taken from Bishop's book



The Bias-Variance Tradeoff

Northeastern University

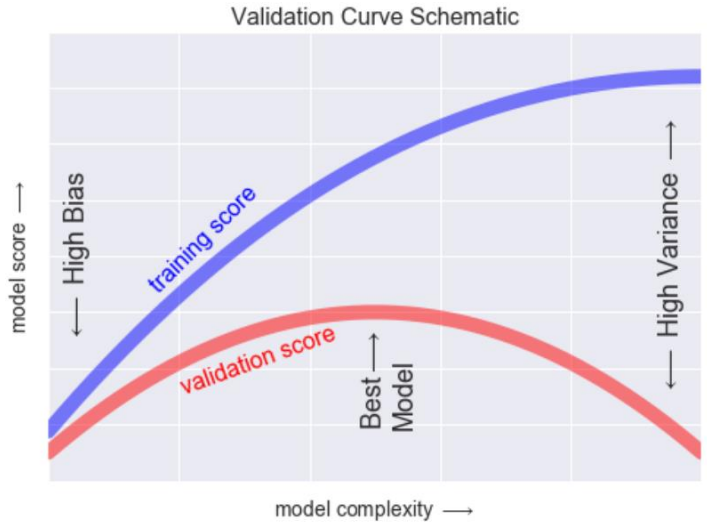
Figure 3.6 Plot of squared bias and variance, together with their sum, corresponding to the results shown in Figure 3.5. Also shown is the average test set error for a test data set size of 1000 points. The minimum value of $(\text{bias})^2 + \text{variance}$ occurs around $\ln \lambda = -0.31$, which is close to the value that gives the minimum error on the test data.



The Bias-Variance Tradeoff

Northeastern University

- We can use the validation curve to find the right balance between bias and variance

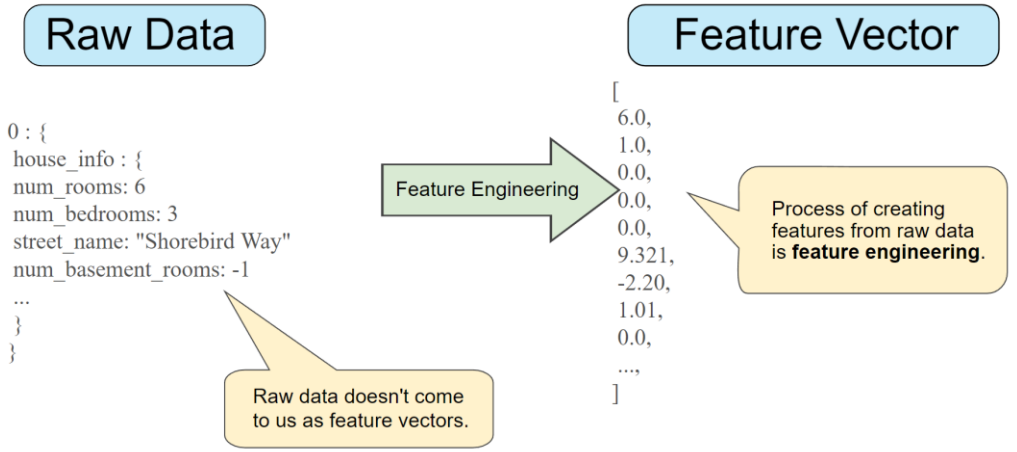


- For high-bias models, the performance of the model on the validation set is similar to the performance on the training set
- For high-variance models, the performance of the model on the validation set is far worse than the performance on the training set

Feature Engineering

Northeastern University

- In the real world, data rarely comes in a tidy, [samples, features] format
- The generation of feature vectors from the raw data is called **feature engineering**

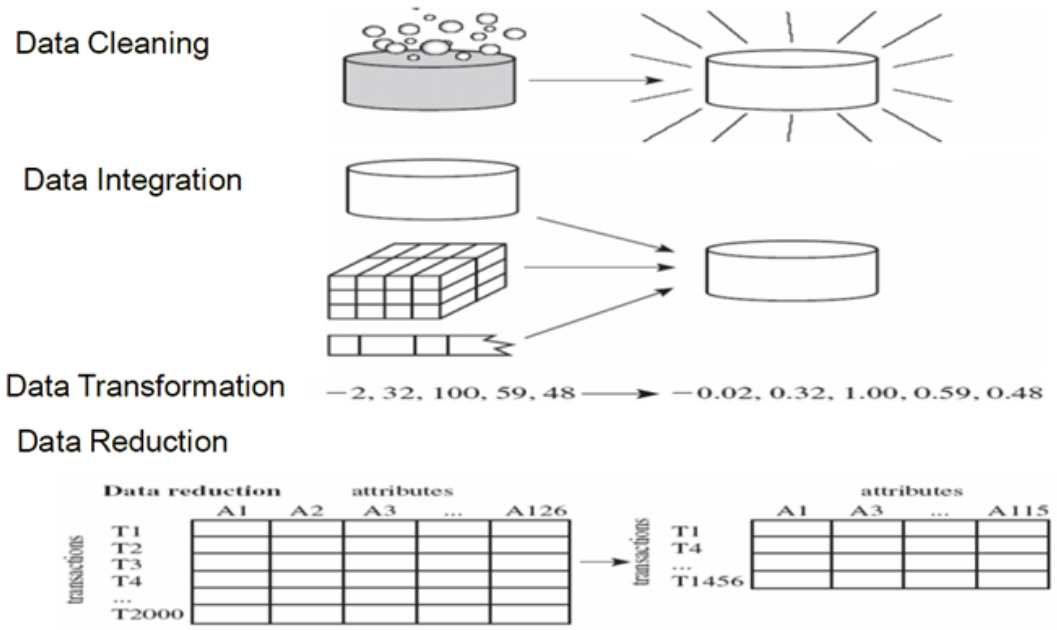


- This process typically involves the following steps:
  - **Data preprocessing:** data preparation, cleaning, and transformation
  - **Feature selection**: selecting the most useful features to train on
  - **Feature extraction**: combining existing features to produce a more useful one

Data Preprocessing

Northeastern University

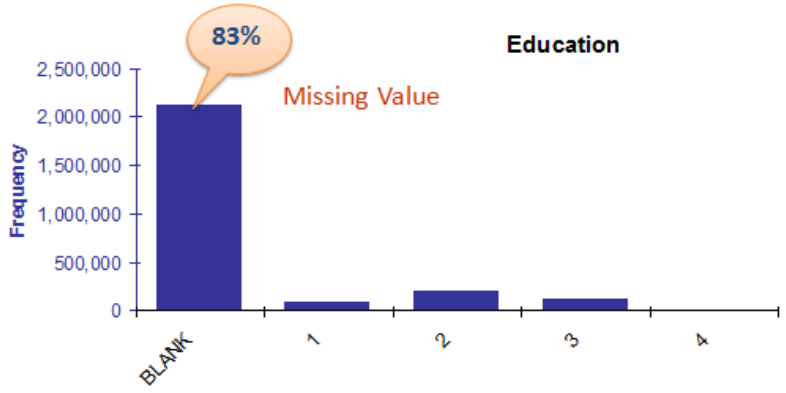
- Handling missing and noisy data
- Data integration
- Transforming categorical data
- Discretization and binarization
- Scaling / normalization
- Outlier detection
- Dimensionality reduction



- **sklearn.preprocessing** includes many methods for data preparation, cleaning, and transformation

Missing Data

- Missing data is one of the most common data quality issues in real-world datasets
- Most data mining algorithms cannot work with missing features
- Missing data may be due to:
  - Information is not collected
    - e.g., people decline to give their age or income level
  - Attributes may not be applicable to all cases
    - e.g., annual income is not applicable to children
  - Equipment malfunction
  - Certain data not considered important at the time of entry



Handling Missing Data

Northeastern University

- Missing data is one of the most common data quality issues in real-world datasets
- Most ML algorithms cannot work with missing features
- Possible solutions:
  - Remove missing values from the dataset
    - If there are many samples with missing values, this is not a viable solution
  - Replace missing values with some appropriate fill value (aka **imputation**)
    - Typically, the mean or median of the feature is used
  - Other, more sophisticated solutions

	col1	col2	col3	col4	col5
0	2	5.0	3.0	6	NaN
1	9	NaN	9.0	0	7.0
2	19	17.0	NaN	9	NaN

mean()

	col1	col2	col3	col4	col5
0	2.0	5.0	3.0	6.0	7.0
1	9.0	11.0	9.0	0.0	7.0
2	19.0	17.0	6.0	9.0	7.0

Imputation of Missing Data

Northeastern University

- The class **SimpleImputer** is an imputation transformer for completing missing values

> *class* `sklearn.impute.` **`SimpleImputer`**(*, *missing_values=nan*, *strategy='mean'*, *fill_value=None*, *verbose=0*, *copy=True*, *add_indicator=False*) [source]

- The **strategy** argument defines the imputation strategy:

Strategy	Description
mean	Replace missing values using the mean along each column. Can only be used with numeric data.
median	Replace missing values using the median along each column. Can only be used with numeric data.
most_frequent	Replace missing values using the most frequent value along each column. Can be used with strings or numeric data.
constant	Replace missing values with fill_value. Can be used with strings or numeric data.

Imputation of Missing Data

Northeastern University

- Example for imputing missing values using the mean:

```
from sklearn.impute import SimpleImputer

imp = SimpleImputer(strategy='mean')
```

```
arr = np.array([[np.nan, 2, 3], [4, np.nan, 6], [10, 5, 9]])
arr
```

```
array([[nan,  2.,  3.],
       [ 4., nan,  6.],
       [10.,  5.,  9.]])
```

```
imp.fit_transform(arr)
```

```
array([[ 7. ,  2. ,  3. ],
       [ 4. ,  3.5,  6. ],
       [10. ,  5. ,  9. ]])
```

Handling Categorical Data

Northeastern University

- A categorical feature represents discrete values that belong to a finite set of categories or classes
- There are two types of categorical variables
  - **Ordinal** features have a natural ordering defined on their values
    - Examples include income level, education level, clothing size, etc.
  - **Nominal** features have no concept of ordering
    - Examples include movie genres, weather names, country names, etc.
- For most ML models categorical features must be first transformed into numbers

Transforming Ordinal Features

Northeastern University

- **OrdinalEncoder** transforms categorical features into integers (ordinal codes)
- The encoding results in a single column of integers [0 to #categories - 1] per feature

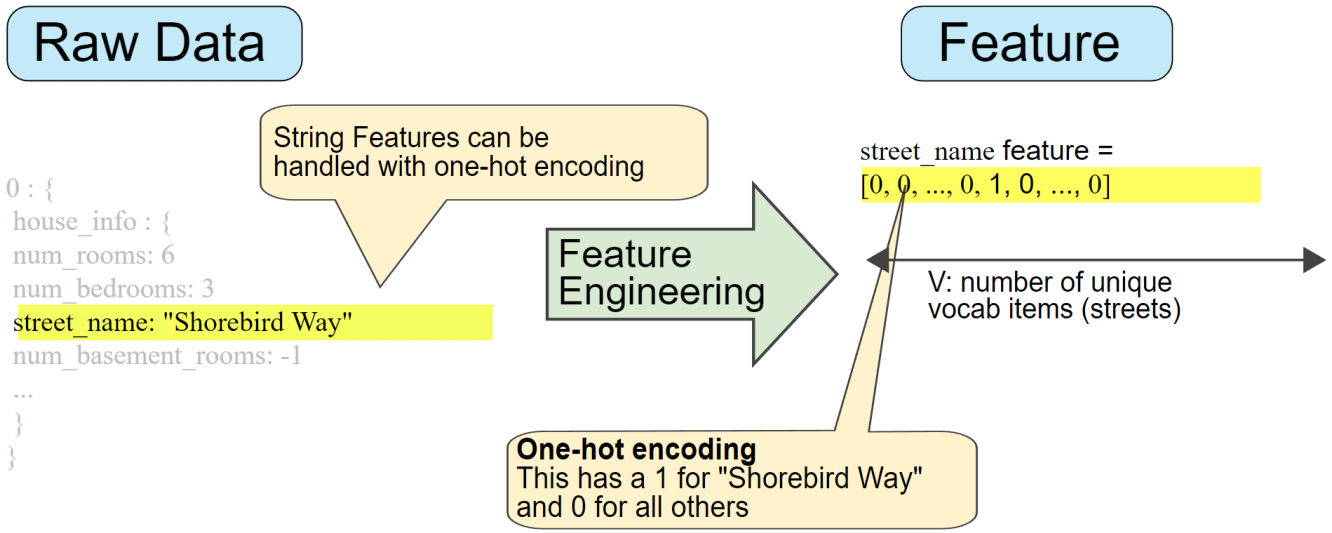
```
from sklearn.preprocessing import OrdinalEncoder

enc = OrdinalEncoder()
```

```
arr = np.array(['MA', 'MA', 'BA', 'PhD', 'BA']).reshape(-1, 1)
enc.fit_transform(arr)
```

```
array([[1.],
       [1.],
       [0.],
       [2.],
       [0.]])
```

One-Hot Encoding

- Convert a categorical feature with *m* labels into a binary vector of size *m* where only one of its elements has 1 and all the other elements have 0



One-Hot Encoding

Northeastern University

- The **OneHotEncoder** class converts categorical values into one-hot vectors

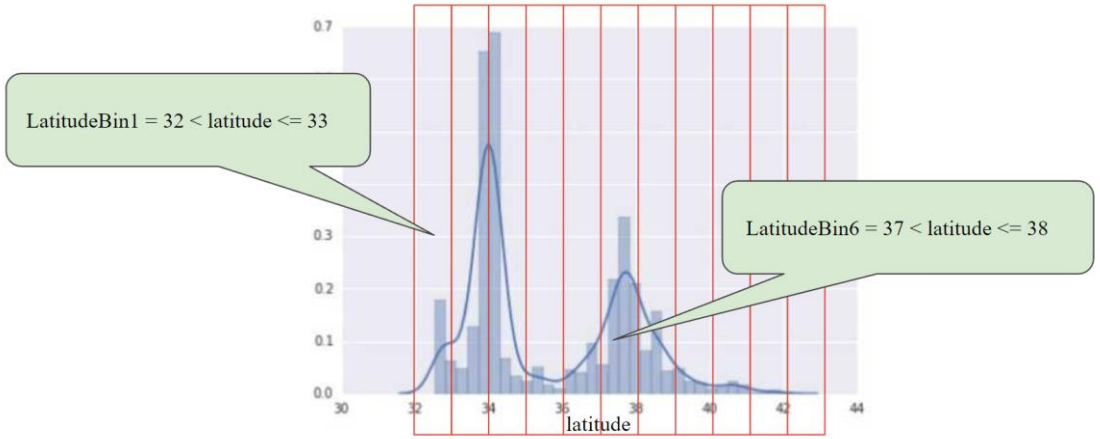
```
from sklearn.preprocessing import OneHotEncoder

enc = OneHotEncoder(sparse=False)
arr = np.array(['Red', 'Green', 'Red', 'Blue', 'Blue']).reshape(-1, 1)
enc.fit_transform(arr)
```

```
array([[0., 0., 1.],
       [0., 1., 0.],
       [0., 0., 1.],
       [1., 0., 0.],
       [1., 0., 0.]])
```

- By default it returns a sparse matrix in which only nonzero values are stored
  - Specify sparse=False to get a dense array

Discretization

- Some ML algorithms require that the features be categorical/binary
- Thus, we often need to **discretize** a continuous attribute into a categorical attribute
- Discretization can also make some real-valued features more meaningful
  - e.g., let's say we need to predict the price of a house based on its location (lat, lng)
  - The price depends on the area where the house is located, not on its exact location



Discretization

Northeastern University

- The result of discretization may be represented as a set of intervals (bins)

$$\{(x_0, x_1], (x_1, x_2], \ldots, (x_{n-1}, x_n)\}$$

- We need to choose:
  - How many bins we want to have
    - Tradeoff: if the bins are too wide, we loose on resolution, but if they are too narrow we increase the risk of overfitting
  - Where to place these bins (where are the split points)
- Two basic approaches:
  - **Equal-width** – all the bins have the same width
  - **Equal frequency (equal depth)** – all the bins have the same number of objects

Discretization

Northeastern University

- **KBinsDiscretizer** can be used to discretize continuous data into intervals

```
class sklearn.preprocessing.KBinsDiscretizer(n_bins=5, *, encode='onehot', strategy='quantile')  [source]
```

  - *n_bins* defines the number of bins to produce (default is 5)
  - *encode* specifies the method used to encode the transformed result
    - Can be ‘onehot’, ‘onehot-dense’, or ‘ordinal’
  - *strategy* defines the strategy used to define the widths of the bins

Strategy	Description
uniform	All bins in each feature have identical widths
quantile	All bins in each feature have the same number of points (default)
kmeans	Values in each bin have the same nearest center of a 1D k-means cluster

Discretization

Northeastern University

```
from sklearn.preprocessing import KBinsDiscretizer

x = np.array([0, 4, 12, 16, 16, 18, 24, 26, 28]).reshape(-1, 1)

dis = KBinsDiscretizer(n_bins=3, strategy='uniform', encode='ordinal')
dis.fit_transform(x)
```

```
array([[0.],
       [0.],
       [1.],
       [1.],
       [1.],
       [1.],
       [2.],
       [2.],
       [2.]])
```

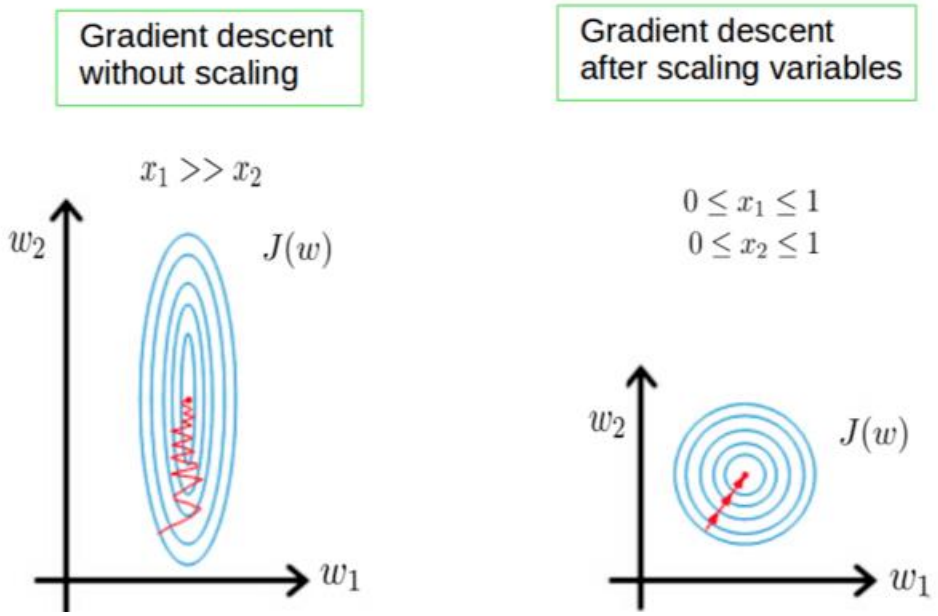
- You can examine the edges of the bins by printing the bin_edges_ attribute:

```
dis.bin_edges_
```

```
array([array([ 0.        ,  9.33333333, 18.66666667, 28.        ])],
      dtype=object)
```

Feature Scaling

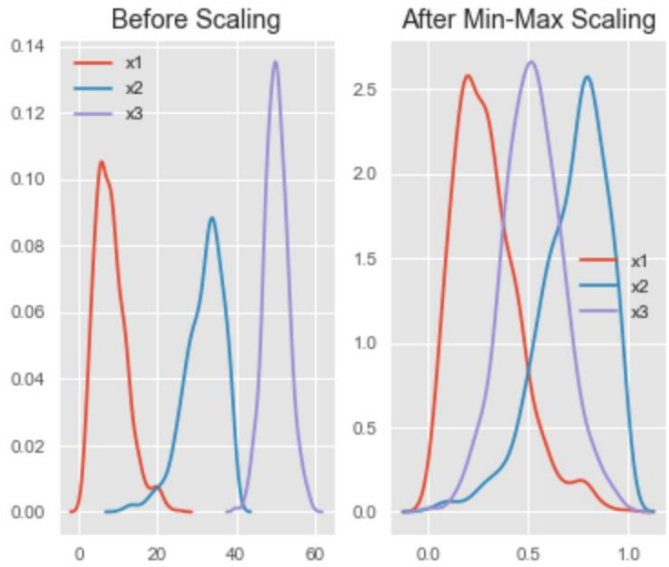
- Many ML algorithms don't perform well when attributes have different scales
  - Models may be biased toward features having higher magnitude values
  - Distance metrics are governed by features with broad range of values
- Feature scaling is used to normalize the range of the features



Min-Max Scaling

- The simplest method
- Rescale the range of the features to [0, 1]

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

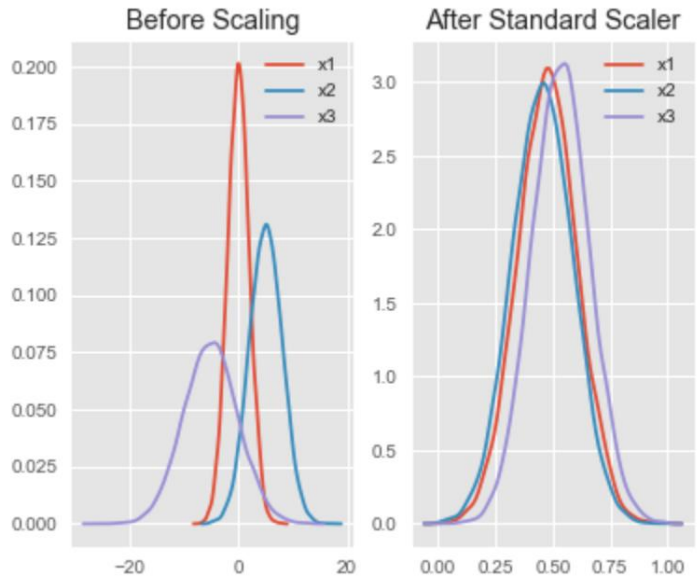


- Sensitive to outliers, since it uses the min and max values

Standard Scaling

- Also known as **Z-score normalization**
- Makes the values of each feature to have zero mean and unit variance

$$x' = \frac{x - \overline{x}}{\sigma}$$



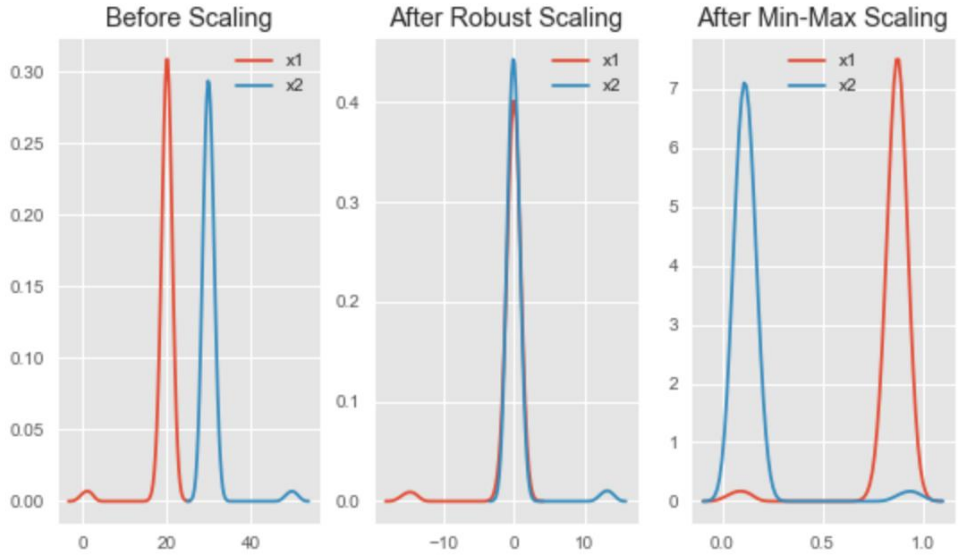
- Also sensitive to outliers, since it uses the mean

Robust Scaling

Northeastern University

- Uses statistics that are robust to outliers

$$x' = \frac{x - \text{median}(x)}{\text{IQR}(x)}$$



- IQR (Inter-Quartile Range) is the difference between the 25th and 75th percentiles

Feature Scaling in Scikit-Learn

Northeastern University

- You can perform feature scaling in Scikit-Learn by using one of the following classes:
  - Min-max scaling

> *class* `sklearn.preprocessing.` **`MinMaxScaler`**(*feature_range=(0, 1), *, copy=True*) [source]

  - Standard scaling

> *class* `sklearn.preprocessing.` **`StandardScaler`**(**, copy=True, with_mean=True, with_std=True*) [source]

  - Robust scaling

> *class* `sklearn.preprocessing.` **`RobustScaler`**(**, with_centering=True, with_scaling=True, quantile_range=(25.0, 75.0), copy=True*) [source]

Feature Scaling in Scikit-Learn

Northeastern University

- Example for using the MinMaxScaler:

```
from sklearn.preprocessing import MinMaxScaler

data = [[1, 1, 1, 1, 1],
        [2, 5, 10, 50, 100],
        [3, 10, 20, 150, 200]]

scaler = MinMaxScaler()
scaler.fit_transform(data)
```

```
array([[0.        , 0.        , 0.        , 0.        , 0.        ],
       [0.5       , 0.44444444, 0.47368421, 0.32885906, 0.49748744],
       [1.        , 1.        , 1.        , 1.        , 1.        ]])
```

Feature Selection

Northeastern University

- Select a subset of relevant features
- Remove **irrelevant features** that contain no information useful for data mining
  - e.g., the student's ID is irrelevant to the task of predicting the student's GPA
- Remove **redundant features** that duplicate much of the information in other attributes
  - e.g., product purchase price and amount of sales tax paid
- Advantages:
  - Reduces training time
  - Reduces dimensionality of data
  - Makes the model easier to interpret

Feature Selection Methods

Northeastern University

- Brute-force approach:
  - Try all possible feature subsets as input to data mining algorithm
- Filter methods:
  - Features are selected before data mining algorithm is run
  - Based on metrics such as correlation and mutual information
- Embedded methods:
  - Feature selection occurs naturally as part of the data mining algorithm
- Wrapper methods:
  - Use the data mining algorithm as a black box to find best subset of attributes

Feature Selection

Northeastern University

- The classes in **sklearn.feature_selection** can be used for feature selection
- For example, **VarianceThreshold** can be used to remove all features whose variance doesn't meet some threshold
- To demonstrate, we'll use the Diagnostic Breast Cancer dataset
  - This dataset contains 569 rows of observations and a total of 30 features

```
from sklearn.datasets import load_breast_cancer

bc_data = load_breast_cancer()
bc_features = pd.DataFrame(bc_data.data,
                           columns=bc_data.feature_names)
bc_features.head()
```

	mean radius	mean texture	mean perimeter	mean area	mean smoothness	mean compactness	mean concavity
0	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.3001
1	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.0869
2	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.1974
3	11.42	20.38	77.58	386.1	0.14250	0.28390	0.2414
4	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.1980

5 rows × 30 columns

Removing Features with Low Variance

Northeastern University

- We can remove features with less than 0.05 variance:

```
from sklearn.feature_selection import VarianceThreshold

vt = VarianceThreshold(threshold=0.05)
vt.fit(bc_features)
```

- To view the variances as well as which features were selected by this algorithm, we can use the variances_ property and the get_support(...) method, respectively:

```
pd.DataFrame({'variance': vt.variances_,
              'select_feature': vt.get_support()},
             index=bc_features.columns).T
```

	mean radius	mean texture	mean perimeter	mean area	mean smoothness	mean compactness	mean concavity	mean concave points	mean symmetry	fr dimer
variance	12.3971	18.4664	589.403	123626	0.000197452	0.00278429	0.00634408	0.00150301	0.000750222	4.97
select_feature	True	True	True	True	False	False	False	False	False	

2 rows × 30 columns

Feature Engineering

Northeastern University

- Feature extraction aims to reduce the number of features by creating new features from the existing ones and then discarding the original features
- For example, assume we have the following data of air pollution by country:

```
pollution = pd.DataFrame([
    {'country': 'New Zealand', 'area': 269190, 'population': 4705818, 'air_pollution': 5},
    {'country': 'Qatar', 'area': 11437, 'population': 2639211, 'air_pollution': 103},
    {'country': 'India', 'area': 2639211, 'population': 1339180127, 'air_pollution': 65},
    {'country': 'Spain', 'area': 504781, 'population': 46354321, 'air_pollution': 9},
    {'country': 'Cameroon', 'area': 475440, 'population': 24053727, 'air_pollution': 62},
    {'country': 'Thailand', 'area': 513120, 'population': 69222885, 'air_pollution': 27},
    {'country': 'Egypt', 'area': 1001449, 'population': 99887475, 'air_pollution': 75}
], columns=['country', 'area', 'population', 'air_pollution'])
pollution = pollution.set_index('country')
pollution
```

country	area	population	air_pollution
New Zealand	269190	4705818	5
Qatar	11437	2639211	103
India	2639211	1339180127	65
Spain	504781	46354321	9
Cameroon	475440	24053727	62
Thailand	513120	69222885	27
Egypt	1001449	99887475	75

```
pollution.corr()['air_pollution']
```

```
area             0.153068
population       0.187176
air_pollution    1.000000
Name: air_pollution, dtype: float64
```

Feature Engineering

Northeastern University

- Let's add a population density feature:

```
pollution['density'] = pollution['population'] / pollution['area']
```

- And now examine the correlation coefficients again:

```
pollution.corr()['air_pollution']
```

```
area             0.153068
population       0.187176
air_pollution    1.000000
density          0.435159
Name: air_pollution, dtype: float64
```

  - There is a relatively high correlation between density and air_pollution

Feature Engineering

Northeastern University

- Here is a small transformer class that adds the density attribute to the features:

```
from sklearn.base import BaseEstimator, TransformerMixin

area_idx, population_idx = 0, 1

class CombinedAttributesAdder(BaseEstimator, TransformerMixin):
    def fit(self, X, y=None):
        return self  # nothing else to do
    def transform(self, X, y=None):
        density = X[:, population_idx] / X[:, area_idx]
        return np.column_stack([X, density])
```

```
attr_adder = CombinedAttributesAdder()
pollution_extra_attrib = attr_adder.transform(X)
pollution_extra_attrib
```

```
array([[2.69190000e+05, 4.70581800e+06, 1.74813998e+01],
       [1.14370000e+04, 2.63921100e+06, 2.30760776e+02],
       [2.63921100e+06, 1.33918013e+09, 5.07416848e+02],
       [           nan, 4.63543210e+07,            nan],
       [4.75440000e+05, 2.40537270e+07, 5.05925606e+01],
       [5.13120000e+05, 6.92228850e+07, 1.34905841e+02],
       [1.00144900e+06, 9.98874750e+07, 9.97429475e+01]])
```